From Uncertainty to Adaptivity: Multiscale Edge Detection and Image Segmentation

By

Kung-Hao Liang

Submitted for the degree of Doctor of Philosophy

to the Higher Degree Committee

University of Warwick

Department of Engineering

University of Warwick, Coventry, U.K.

August 1997

Contents

List of Abbreviations

ATR: The approximate-true ratio.

EHF: The energy of the high-frequency components.

FCR: The false-correct ratio.

FT: Fourier transform.

MAP: Maximum *a posteriori*.

MFT: Multiresolution Fourier transform.

MRCBS: The multiscale edge detection scheme using the regularised cubic B-spline fitting.

MRF: Markov Random Field.

NSD: The standard deviation of the Gaussian noise.

OFC: Optical Flow Constraint.

PCS: Percentage of the Correct Segmentation.

RCBS: Regularised cubic B-spline.

TEHF: The threshold on the energy of the high-frequency components.

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. T. Tjahjadi, for all his supports and advices throughout these three years. Without his patient proof reading and his comments, this thesis would not appear in such a well-written form. The financial support from the Linda Sumartini Foundation is gratefully acknowledged.

I would like to thank Dr. R.Staunton for his help while Dr. Tjahjadi was on sabbatical leave. I would also like to thank Chang-Tsun Li for all the stimulating discussions.

Finally, I would like to thank my parents, my sister and my dearly loved Yun for their supports and encouragements. They are the major motivations and driving forces for me to achieve the PhD degree.

Declaration

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy by the Higher Degree Committee at the University of Warwick. The thesis has been composed and written by myself based on the research undertaken by myself. The research materials have not been submitted in any previous application for a higher degree. All sources of information are specifically acknowledged in the content.

Kung-Hao Liang

Summary

This thesis presents the research on two different tasks in computer vision: edge detection and image segmentation (including texture segmentation and motion field segmentation). The central issue of this thesis is the uncertainty of the joint space-frequency image analysis, which motivates the design of the adaptive multiscale/multiresolution schemes for edge detection and image segmentation. Edge detectors capture most of the local features in an image, including the object boundaries and the details of surface textures. Apart from these edge features, the region properties of surface textures and motion fields are also important for segmenting an image into disjoint regions. The major theoretical achievements of this thesis are twofold. First, a scale parameter for the local processing of an image (e.g. edge detection) is proposed. The corresponding edge behaviour in the scale space, referred to as Bounded Diffusion, is the basis of a multiscale edge detector where the scale is adjusted adaptively according to the local noise level. Second, an adaptive multiresolution clustering scheme is proposed for texture segmentation (referred to as Texture Focusing) and motion field segmentation. In this scheme, the central regions of homogeneous textures (motion fields) are analysed using coarse resolutions so as to achieve a better estimation of the textural content (optical flow), and the border region of a texture (motion field) is analysed using fine resolutions so as to achieve a better estimation of the boundary between textures (moving objects). Both of the above two achievements are the logical consequences of the uncertainty principle. Four algorithms, including a roof edge detector, a multiscale step edge detector, a texture segmentation scheme and a motion field segmentation scheme are proposed to address various aspects of edge detection and image segmentation. These algorithms have been implemented and extensively evaluated.

Chapter 1

Introduction

In recent years, digital images have been used extensively in medical, industrial and telecommunication applications due to the rapid progress of digital techniques. Various image modality, including charge-coupled devices for light and thermal imaging, laser range imaging, Synthetic Aperture Radar imaging (SAR), as well as various medical imaging modalities such as the Magnetic Resonance imaging (MRI), the Computed Tomography (CT) and the Positron Emission Tomography (PET), have been developed to generate 2-D and 3-D images using signals with lower dimensions [18]. At the same time, various image processing techniques have been investigated to extract useful information from digital images, which normally contain some physical defects such as noise or blurring. The research discipline of image processing includes computed imaging, filtering, restoration and coding, *etc.* The discipline of computer vision, which is closely related to image processing, emerged in the 70s. Computer vision starts with the physiological and psychophysical investigations of the human (mammalian) visual systems, and then constructs artificial visual systems using sensors together with various computing algorithms [67]. The basic principle behind computer vision is that the visual process is a computational process, therefore a systematic investigation in vision can result in a series of algorithms

which build up an artificial visual system. Tasks of computer vision include structure from motion, shape from shading, object tracking, *etc.* [40].

Although image processing groups and computer vision groups have slightly different emphases and attitudes toward research (the former tend to devise generic approaches, whereas the later are more interested in the interpretation of physical scenes from images), their close relationship cross fertilises each other. For example, segmentation and feature extraction are normally the starting points of both fields. In addition, the adaptivity in scale is generally the key component for an efficient computer vision or image processing algorithm.

Computer vision is an inverse process because the physical scene is reconstructed through images [6]. Since the physical information of a scene is incomplete in the image, assumptions have to be made during the reconstruction stage to estimate the missing information. An inverse problem is ill-posed in the presence of noise, violating the three requirements of well-posedness: (1) a solution exists, (2) the solution is unique, and (3) the solution depends continuously on the input data (see Section 3.2). Regularisation is a procedure for formulating a well-posed task by employing assumptions which guarantee a unique and stable solution to the task (see Section 3.2). It has been pointed out that regularisation has a close relationship with the concept of scale [94]. This close relationship leads to the Bounded Diffusion theory presented in Chapter 4, which is one of the central issues addressed in this thesis.

In the early 80's, Marr proposed a hierarchy of tasks which lead to the ultimate goal of computer vision. In this hierarchy, the high level information (i.e. symbolic descriptions of the physical scene) is derived from the middle level (e.g. $2\frac{1}{2}D$ surface maps, object shapes or movements) and the low level (i.e. primal sketches) information [67]. Marr's systematic approach toward vision became a standard paradigm, where tasks in different levels are

tackled separately. In this thesis the tasks of edge detection and image segmentation using textures and motion fields are investigated. These tasks are low-level tasks, which determine the performance of the subsequent high-level tasks.

1.1 Edge Detection

Identifying and locating object boundaries in an image is an essential task in low-level computer vision because an object boundary provides an initial description of the scene. Object boundaries manifest themselves as significant discontinuities between image grey-levels of adjacent pixels, thus, the detection of these discontinuities (referred to as edges) attracts enormous attention from computer vision groups. A step edge corresponds to an abrupt change in grey-level (i.e. a discontinuity), whereas a roof edge corresponds to the first order discontinuity in the image gradients. Marr uses edges to illustrate the concept of primal sketches [66], therefore the performance of the higher level computer vision relies on the accurate detection of edges. Physiological evidences also show that the recognition of edges plays an important role in mammalian vision. For example, Hubel and Wiesel have shown that mammalian cortex contains a population of feature detectors which is tuned to edges and bars of various width and orientation [66].

The role of an edge detector is to locate the discontinuities in image grey levels accurately in the presence of noise. This is a dilemma because a small degree of smoothing is preferable for locating the edges, whereas a large degree of smoothing is required for suppressing noise. This dilemma is commonly acknowledged as the uncertainty principle (see Section 1.4) because a large operator kernel provides a large degree of smoothing, which inevitably reduces the resolution of the edge maps (e.g. [105]). Contrary to this common belief, Chapter 4 demonstrates a new concept of scale for local image processing

tasks such as edge detection. This new concept is referred to as the Bounded Diffusion theory, where the adjustment of the regularisation effect according to the noise levels does not increase the size of the operator kernel.

1.2 Texture Segmentation

Image segmentation is the process by which an image is partitioned into disjoint regions, each of which has a homogeneous region property such as a texture or a motion field. It is an intermediate process toward a high-level interpretation of a scene. Although object boundaries manifest themselves as edges, edge detection alone cannot serve as a complete image segmentation scheme. The reason being that edge detectors capture both boundaries and surface textures of an object. In addition, a procedure of edge linking is required after edge detection to compose a parametric contour (i.e. a chain of boundary locations) to complete the segmentation process (see Section 2.3). The difficulties of image segmentation is normally under-estimated because our natural aptitude to interpret a visual scene is excellent and spontaneous [105]. There are two approaches to tackle the problem of image segmentation: to consider region properties such as textures or colours; and to use information from multiple frames such as stereos or motion fields to recognise occlusion. In this thesis, texture segmentation and motion field segmentation are investigated.

Texture segmentation partitions an image according to the pattern of variations of the image grey levels. In texture segmentation, the boundary of a region with homogeneous texture has to be accurately located, while the texture content within the boundary is used to determine the homogeneity. The former requires a small operator kernel to achieve high spatial resolution, while the latter requires a large kernel to achieve high feature resolution.

This is a dilemma occurring in the joint space-frequency analysis, i.e. the uncertainty principle (see Section 1.4), which indicates that a good texture segmentation can only be achieved using an adaptive multiscale (multiresolution) approach.

1.3 Motion Field Segmentation

The optical flow constraint equation (OFC) proposed by Horn and Schunck [43] is commonly used to derive the motion field from image sequences. OFC assumes that the image grey level of a moving point is stationary with respect to time, thus

$$\frac{dS(x,y,t)}{dt} = \frac{\partial S(x,y,t)}{\partial x}\frac{dx}{dt} + \frac{\partial S(x,y,t)}{\partial y}\frac{dy}{dt} + \frac{\partial S(x,y,t)}{\partial t} = 0, \tag{1.1}$$

where $S(x,y,t)$ is the grey-level at an image pixel (x,y) at time t; and $u = \frac{dx}{dt}$ and $v = \frac{dy}{dt}$ represent the two orthogonal components of the velocity of the image pixel.

Deriving the optical flow from OFC's is an ill-posed task because the two variables of u and v are determined using a single constraint equation. This is commonly known as the aperture problem. In addition, OFC is derived under the assumption that $S(x,y,t)$ is continuous, thus errors of the optical flow will be introduced at positions with grey-level discontinuities usually caused by the occlusion of objects. The above two problems can be solved to a certain extent by the regularisation technique. For example, the multi-point approach assumes the optical flow of adjacent image pixels are identical (i.e. the regularisation), thus a smoothed but unique optical flow field is determined under this assumption. This approach still suffers from the dilemma of uncertainty because the solutions of OFC's tend to be erroneous due to the ill-conditioness (Section 6.1) when a small window (i.e. high spatial resolution) is used. If a large window is used (i.e. low spatial resolution), then the optical flow is an averaged value within the window, which is less likely to be influenced by noise but is more likely to contain several objects,

each of which corresponds to a distinct motion field in the image. Thus, an adaptive multiresolution scheme is required to circumvent the uncertainty.

1.4 Uncertainty, Scale and Multiresolution

As indicated in the previous three sections, researches on edge detection, texture segmentation and motion field segmentation all involve (by their nature or by a common misunderstanding among researchers) the uncertainty of the joint space-frequency analysis. In this section, the uncertainty principle is examined in detail. Let $h(x)$ denote the operator kernel (a real function with unit L_2 norm) of the analysis, and $H(w)$ be the Fourier Transform of $h(x)$. Assume $h(x) \rightarrow 0$ when $x \rightarrow \pm\infty$. The spatial resolution $\triangle x$ and the feature resolution $\triangle w$ in the time-frequency plane (also referred to as a spectrogram [84]) are defined as the variances of $h(x)$ and $H(w)$ respectively [20, 98]:

$$\triangle x^2 = \int_{-\infty}^{\infty} x^2 \left|h(x)\right|^2 dx,$$

$$\begin{aligned} \triangle w^2 &= \int_{-\infty}^{\infty} w^2 \left|H(w)\right|^2 dw \\ &= \int_{-\infty}^{\infty} \left|h'(x)\right|^2 dx. \end{aligned}$$

Thus,

$$\begin{aligned} \triangle x^2 \triangle w^2 &= \int_{-\infty}^{\infty} x^2 \left|h(x)\right|^2 dx \times \int_{-\infty}^{\infty} \left|h'(x)\right|^2 dx \\ &\geq \left|\int_{-\infty}^{\infty} x h(x) h'(x) dx\right|^2. \qquad (Schwarz\ \ inequality) \end{aligned} \tag{1.2}$$

Since

$$\begin{aligned} \int_{-\infty}^{\infty} x h(x) h'(x) dx &= \frac{1}{2} \int_{-\infty}^{\infty} x dh(x)^2 \\ &= \frac{1}{2} \left[x h(x)^2\right]_{-\infty}^{\infty} - \frac{1}{2} \int_{-\infty}^{\infty} h(x)^2 dx \\ &= -\frac{1}{2}, \end{aligned}$$

equation (1.2) becomes

$$\Delta x^2 \, \Delta w^2 \geq \frac{1}{4}. \tag{1.3}$$

The above formula imposes a lower bound on the time-frequency product, which shows that high resolutions in both the spatial domain and the frequency domain cannot be achieved simultaneously in the joint space-frequency analysis. The equality of (1.3) is reached (i.e. the uncertainty of the time-frequency product is minimised) when the two functions involved in the Schwarz inequality of (1.2) are proportional to each other, i.e. [20, 98]

$$\frac{h'(x)}{xh(x)} = constant,$$

where the solution of $h(x)$ is a Gaussian.

The issue of uncertainty, scale and resolution is the central issue of this thesis. It has also been discussed in scale-space filtering methods (e.g. [59, 108]), wavelet methods (e.g. [64, 84]), diffusion methods (e.g. [79, 102, 103]), as well as various multi-scale/multiresolution techniques (e.g. [5, 62, 63, 110]). The underlying objectives of these theories are similar, which are the decomposition of an image into different spatial frequency channels (scales) so as to facilitate the joint space-frequency or space-scale analysis. Generally, a scale is defined as the standard deviation of the Gaussian pre-filter σ (e.g. [5, 62]), a continuous parameter, which indicates the degree of smoothing of an image.

The standard deviation σ of the Gaussian is proportional to the spatial resolution Δx. In the literature, the spatial extent of the block-shaped window in the quad-tree image structure is also referred to as the resolution, which is a dyadic series. In this thesis the terminology of the scale is used in continuous situations, whereas the resolution is used to indicate the block size of the quad-tree.

Since a scale reflects the degree of smoothing, it is adjusted in accordance with the

noise level. If an image is noise-free, then the inner scale (i.e. the smallest scale which is determined by the granularity of the image) always provides the most complete information of an image. Otherwise, an adequate scale is required to suppress noise. An accurate modelling of the noise level is thus essential for the indication of the scale. This will be discussed in Section 4.3.1.

1.5 Organisation of the Thesis

This thesis tackles three computer vision tasks of edge detection, texture segmentation and motion field segmentation. In this thesis, uncertainty, scale and adaptivity are the central concepts which are closely linked together. Due to the local nature of edges (see Section 1.1), Bounded Diffusion is proposed to provide a local scale factor for edge detection, where the spatial extent of the operator kernel is independent of the scale. Texture segmentation and motion field segmentation are susceptible to uncertainty, thus an adaptive multiresolution clustering method is devised to circumvent the uncertainty.

The organisation of the thesis is as follows: Chapter 2 reviews the important edge detectors to illustrate an evolution of concepts for edge detection. Chapter 3 presents the theories of regularisation and well-posedness, as well as a design of a roof edge detector. In Chapter 4, a local scale factor of α is proposed as the basis for a multiscale edge detector, where the scale is adjusted adaptively according to the local noise level. Chapter 5 presents a multiscale texture segmentation scheme, referred to as Texture Focusing, where the regional resolutions are adaptively determined according to the boundary of the texture. In Chapter 6, Texture Focusing is incorporated with the optical flow multi-point method to achieve the motion field segmentation. Finally, Chapter 7 concludes this thesis.

Chapter 2

Edge Detection

2.1 Finite Difference Edge Detector

A digital image is a two-dimensional (2-D) array of grey levels, which correspond to the sampled light intensities in light images, or the depth values in range images. Normally the grey levels are indicated by a third coordinate which is perpendicular to the 2-D image plane. Therefore, the digital image $g(x_i, y_i)$ is viewed as the sampled data from a 3-D continuous surface $f(x, y)$, where x and y are spatial coordinates, and x_i and y_i are the grid points of x and y.

As defined in Section 1.1, a step edge corresponds to an abrupt change in grey level, whereas a roof edge corresponds to the discontinuities in image gradients. The abrupt change occurs in grey-level $g(x_i, y_i)$ when the underlying surface $f(x, y)$ is very steep. A step edge is thus defined as the set of pixels (x, y) where the gradient of $f(x, y)$ exceeds a certain threshold value:

$$\sqrt{\frac{\partial f(x,y)}{\partial x} + \frac{\partial f(x,y)}{\partial y}} \geq threshold.$$

In a sampled image, the derivatives of $\frac{\partial f(x,y)}{\partial x}$ and $\frac{\partial f(x,y)}{\partial y}$ are approximated using the finite

difference method, i.e.

$$\frac{\partial f(x,y)}{\partial x} \approx g(x_{i+1}, y_i) - g(x_i, y_i),$$
$$\frac{\partial f(x,y)}{\partial y} \approx g(x_i, y_{i+1}) - g(x_i, y_i). \tag{2.1}$$

Alternatively, they can be determined by convolving a weighting mask (also known as a window or an operator kernel) with the image. These masks are square matrices, e.g.

$$\frac{\partial f(x,y)}{\partial x} \approx g(x_i, y_i) * \begin{bmatrix} -1 & -k & -1 \\ 0 & 0 & 0 \\ 1 & k & 1 \end{bmatrix},$$

$$\frac{\partial f(x,y)}{\partial y} \approx g(x_i, y_i) * \begin{bmatrix} -1 & 0 & 1 \\ -k & 0 & k \\ -1 & 0 & 1 \end{bmatrix},$$

where $*$ denotes convolution; k is a positive constant. Compared with the finite difference method of equation (2.1), these weighting masks introduce a smoothing effect along the orientation which is perpendicular to the derivative so as to suppress noise. Different value of k have been proposed heuristically. For example, $k = 2$ in the Sobel operator and $k = 1$ in the Prewitt operator [30]. Masks with different sizes are also proposed for different degrees of smoothing. However, these approaches are incapable of calibrating the size of the masks according to the noise level so as to avoid excessive bluring.

2.2 Laplacian of Gaussian Operator

The finite difference approach which thresholds the image gradient produces thick edges in the edge map. This is because more than one pixel adjacent to the grey-level discontinuity qualify as edge pixels due to their large gradients. To produce one-pixel wide edge sketches, the pixels which are the local maximum of the gradient are thus defined as edges. These

pixels correspond to the zero values of the second order derivative along the gradient in the underlying surface f, i.e.

$$\frac{\partial^2 f}{\partial n^2},$$

where n is the coordinate along the gradient [94]. Normally the above process is simplified to detecting the zero-crossings of the Laplacian of the image, i.e. $\nabla^2 f$, as an edge.

The detection of an edge using the Laplacian operator or the finite difference operator is an ill-posed task because the inherent differentiation enhances the high-frequency components of the image including noise. The results are thus highly noise-sensitive. A rigorous definition of well-posedness is presented in Section 3.2. The method of regularisation converts an ill-posed task into well-posed. Torre and Poggio showed that regularisation can be achieved by convolving the data with a cubic-spline filter, which has a shape similar to a Gaussian [94].

Marr and Hildreth [66] proposed the Laplacian of Gaussian operator (LoG), an isotropic operator, for edge detection. The idea is to convolve an image with a Gaussian smoothing pre-filter $G(\sigma, x, y)$ for the regularisation, and then calculate the Laplacian of the smoothed image to produce an edge-enhanced image $EE(x_i, y_i)$ (a band-passed image), i.e.

$$EE(x_i, y_i) = \nabla^2 \left[G(\sigma, x, y) * g(x_i, y_i)\right] = \left[\nabla^2 G(\sigma, x, y)\right] * g(x_i, y_i),$$

where σ is the standard deviation of the Gaussian. The LoG operator $\nabla^2 G(\sigma, x, y)$ is

$$\nabla^2 G(\sigma, x, y) = \frac{x^2 + y^2 - 2\alpha^2}{2\pi\alpha^6} e^{\frac{-(x^2+y^2)}{2\alpha^2}},$$

which is also known as the "Mexican hat" operator due to its shape [66]. Note that the 2-D Gaussian can be decomposed to two 1-D Gaussians, i.e.

$$G(\sigma, x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-x^2}{2\sigma^2}} \times \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-y^2}{2\sigma^2}} = G(\sigma, x, 0) \times G(\sigma, 0, y).$$

Similarly

$$\begin{aligned}\nabla^2 G(\sigma, x, y) &= (\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2})[G(\sigma, x, 0) \times G(\sigma, 0, y)] \\ &= \frac{\partial^2}{\partial x^2} G(\sigma, x, 0) \times G(\sigma, 0, y) + G(\sigma, x, 0) \times \frac{\partial^2}{\partial y^2} G(\sigma, 0, y).\end{aligned}$$

This shows that the 2-D convolution of an image with $\nabla^2 G(\sigma, x, y)$ can be simplified as four 1-D convolutions. The LoG operator can also be approximated by the Difference of Gaussian (DoG) kernel [66], which is the basic principle of the Laplacian pyramid [13].

Figure 2.1 shows the image of "Trevor" and the edge-enhanced image obtained from the convolution of "Trevor" with the LoG kernel. In Marr and Hildreth's approach, the edge map is further derived using the set of pixels with zero values in the edge-enhanced image.

(a) (b)

Figure 2.1: (a) The image of "Trevor"; (b) The edge-enhanced image produced by the LoG filter.

The value of σ in the Gaussian pre-filtering corresponds to the degree of smoothing. When σ is small, all the tiny edges with small gradients are extracted. This is why the LoG edge detector also incorporates a thresholding process on the gradient to separate salient edges from tiny edges [36]. Another way to deal with the tiny edges is to introduce

a large amount of blurring by increasing the value of σ, which however causes the edges to be displaced from its original positions (see Chapter 4). Marr and Hildreth have discussed the edge behaviour under various value of σ [66]. This concept is further developed by Witkin as the scale-space theory [108].

2.2.1 Uncertainty and Gaussian

Canny employed the numerical optimisation method to design an optimised 1-D kernel for an edge detector [16]. Three criteria, i.e. the suppression of noise, the localisation of edges, and one single response to an edge, are chosen for the optimisation. The result showed that the optimised kernels for step and roof edges are approximated by the first and the second order derivatives of a Gaussian respectively. Canny's first two criteria correspond to the uncertainty which occurs in edge detection, i.e. a large degree of smoothing is required to suppress noise, whereas a small degree of smoothing is preferable for locating the edges accurately. Thus Canny's numerical result confirms the theoretical prediction of the uncertainty principle that the Gaussian is the optimum kernel (see Section 1.4).

Canny also suggested the use of the $\frac{\partial^2}{\partial n^2}G(\sigma, n)$ operator instead of the LoG operator for edge detection, where n is the coordinate along the gradient. This agrees with the natural definition of an edge, i.e. $\frac{\partial^2 f}{\partial n^2}$, which is also accepted by Torre and Poggio [94], Haralick (see Section 2.3), and the MRCBS proposed in Chapter 4.

The fact that the optimised 1-D roof edge detector is the second order derivative of the Gaussian suggests that the Laplacian kernel, the sum of two 1-D second order derivatives in the orthogonal directions of an image, is a candidate for roof edge detection. In Chapter 3 the second order derivatives are used to indicate the presence of roof edges, where the Gaussian smoothing is achieved by a regularised fitting process.

2.3 Surface Fitting Edge Detector

Reconstructing the physical scene from digital images is the objective of computer vision. Under this premise, an intuitive procedure is to fit the sampled grey levels with a continuous surface. Haralick proposed the facet-model based edge detector [39], which comprises two steps. First, a set of 2-D Chebychev orthogonal polynomials are employed to fit a window of image pixels g. The reconstructed underlying surface f, referred to as a facet, is a linear combination of these Chebychev polynomials, where a series of coefficients c determine the weights of the polynomial. These coefficients are determined by the least-square fitting, which minimises a well-posed quadratic energy function $E = \| Ac - g \|^2$, where $\| \cdot \|$ denotes the l_2-norm, and matrix A maps the coefficients from c space to g space. Hence, the minimum of E occurs when $c = (A^T A)^{-1} A^T g$ [90], which indicates that c is continuously dependent on g. The least-square fitting provides a smoothing effect similar to the result of the Gaussian pre-filtering.

Second, the gradient of the facet f and the second derivative along the gradient orientation (i.e. $\frac{\partial^2 f}{\partial n^2}$) are determined. If a zero-crossing occurs on $\frac{\partial^2 f}{\partial n^2}$ at a pixel, and the gradient of the pixel is larger than a threshold, then this pixel is classified as an edge pixel. Figure 2.2(b) shows the edge map produced by the Haralick edge detector.

Apart from the Haralick operator which classifies edge pixels according to the derivatives of the local facet, Nalwa and Binford argued that an edge is a piecewise curve composed of short, linear edge elements referred to as edgels, each of which is characterised by a position and a direction [75]. The 1-D surface (i.e. the surface where the grey-level is constant in one spatial direction) is thus used to match the local surface of an edgel. The hyperbolic tangent function ($tanh$) is chosen as the 1-D surface because it is similar in shape to a step edge. This method is summarised as follows:

(a) (b)

Figure 2.2: (a) The image of "Trevor"; (b) The edge map produced by the Haralick edge detector (the threshold is 4).

1. A 1-D plane is used to least-square-fit a window of image pixels;
2. Determine the gradient orientation of the fitted plane;
3. A third-order 1-D polynomial is used to least-square-fit the same pixels to refine the estimation of the gradient;
4. A 1-D *tanh* function is used to least-square-fit the same window;
5. A quadratic polynomial is used to least-square-fit the image pixels along the orientation determined at step 3;
6. Compare the error of the least-square-fitting in steps 4 and 5. If the error in 4 is greater than in 5, then an edgel is determined. The 1-D tanh function is thus used to determine the orientation and the magnitude of the edgel.

Nalwa and Binford argued that their edgel detector is more useful than edge detectors because the result produced by an edgel detector is more suitable for the subsequent linking process.

2.4 Active Contour Model

The active contour scheme is devised to extract a contour directly from the image grey levels [50]. This scheme minimises the functional E:

$$E = \int_0^1 E_{intern}(\vec{v}(s)) + \int_0^1 E_{image}(\vec{v}(s))ds + \int_0^1 E_{const}(\vec{v}(s))ds,$$

where the contour is represented parametrically by $\vec{v}(s) = (x(s), y(s))$, with arc length parameter $s \in [0, 1]$. E_{intern} is the internal energy due to the elastic deformation and the bending of the contour. E_{image} corresponds to the target feature such as a step edge or a roof edge. The external constant term E_{const} is defined as the distance between the contour and a given spatial position. This is to facilitate the man-machine interaction. The solution $\vec{v}(s)$ is determined by minimising E. For example, the energy functional of a roof-edge contour is:

$$E = \int (\alpha(s)|\vec{v}_s(s)|^2 + \beta(s)|\vec{v}_{ss}(s)|^2)ds + \sum_s g(v(s))$$

where $\alpha(s)$ and $\beta(s)$ are weightings; $\vec{v}_s(s)$ and $\vec{v}_{ss}(s)$ are, respectively, the first and the second order derivatives of the contours $\vec{v}(s)$. $g(\vec{v}(s))$ denotes the grey level along $\vec{v}(s)$.

A few nodes, referred to as snaxels, have to be specified in the active contour scheme to provide an initial condition. This is because the contours are composed of curves (e.g. B-Splines) interpolated between the snaxels. The number of snaxels thus reflects the resolution of the contour. However, these snaxels have to be specified individually, which is a very tedious work. Therefore, Schnabel extended the active contour model to a multiscale contour extraction scheme of shape focusing [87], which starts from an image blurred by Gaussian with a coarse scale (i.e. a large value of σ) where fewer snaxels are required. A rough contour is thus obtained by the active contour scheme. This contour serves as the initial condition for the following steps in the finer scales where more snaxels

are inserted automatically. Thus, a contour with a complex shape is gradually extracted as the scale decreases.

2.5 Discussion

This chapter reviews the advantages and drawbacks of the finite difference operators, the Laplacian of Gaussian operator, the Haralick's facet model, the *tanh* 1-D surface fitting approach, as well as the active contour model for edge detection. The use of the finite difference operators for edge detection is intuitive but ill-posed, thus it fails on noisy images. The Laplacian of Gaussian operator, which combines the Gaussian pre-filter and the Laplacian differentiator, is well-posed. However, the isotropic Laplacian of ∇^2 is an approximation of the true edge differentiator of $\frac{\partial^2}{\partial n^2}$ [94]. The Haralick operator comprises the facet model (for a local surface fitting) and the $\frac{\partial^2}{\partial n^2}$ differentiator. Nalwa and Binford proposed a complex approach which is based on the 1-D *tanh* surface fitting. The local surface around an edge is fitted and therefore, the parameters of the edgels are available for the subsequent edge linking process. The active contour scheme, which is based on the minimisation of a regularisation functional, produces a contour directly from the grey level image according to the required features (e.g. step or roof edges). However, a few snaxels have to be specified to serve as the initial condition.

Both the Haralick scheme and the LoG scheme are well-posed edge detectors. Even though the theoretical analysis shows that the $\frac{\partial^2}{\partial n^2}$ differentiator (as in the Haralick scheme) is preferable to ∇^2 (as in the LoG scheme), the experimental results show that the performance of the LoG operator is comparable with the Haralick operator [36]. The Haralick scheme, where the Gaussian pre-filter is not used, lacks a systematic approach to the adjustment of the degree of smoothing, which is achieved by the least-square fitting.

In contrast, the standard deviation of the Gaussian provides a scale parameter for the scale-space theory [108] as well as various multiresolution schemes such as the Laplacian pyramid [13]. Gaussian filtering also facilitates the multiscale active contour method [87]. A detailed discussion on multiscale edge detection will be presented in Chapter 4.

Chapter 3

Roof Edge Detection and Regularisation

3.1 Roof Edge Detection

A roof edge is an important feature in various applications. For example, it is argued that human facial expressions in images are better depicted by roof edges than by step edges [78]. The extraction of roof edges from digital terrain models plays an important role in lithology, structural geology and geo-morphology [83]. Roof edges are also important in the analysis of aero-magnetic images [44] and the segmentation of range images [41, 77].

A roof edge is generally defined as a discontinuity in the first order derivative of a 1-D grey-level profile f [77, 54]. This definition is adopted in this thesis. However, in certain instances a sign change in the first order derivatives on the two sides of the discontinuity is also required [38], i.e. $f'(t_o+) \times f'(t_o-) \leq 0$, where the discontinuity occurs at the position t_o. Thus, a roof edge is a local maximum or a local minimum of a grey-level profile, which are referred to as a ridge or a valley, respectively, in the literature [38, 44, 83]. Figure 3.1 illustrates an example of a roof edge occurring at the origin of the 1-D coordinate along

the principle orientation (the principle orientation is defined in Section 3.4.1).

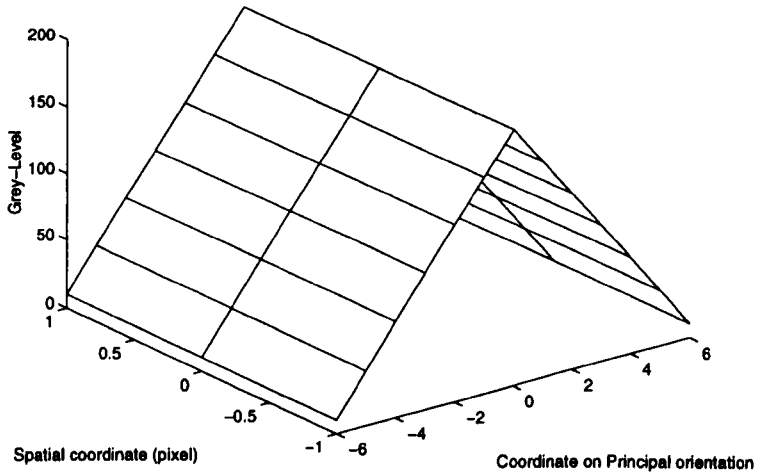
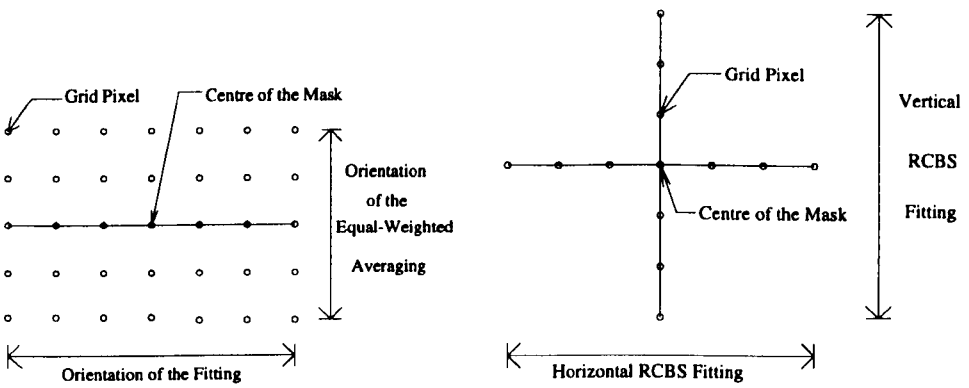


Figure 3.1: An example of a roof edge.

Various schemes for detecting roof edges (or ridges and valleys) have been proposed. Pearson and Robinson proposed a valley detector, which uses a set of criteria to examine the relative grey levels of a group of neighbouring pixels [78]. These criteria determine the local extremum of the second order derivative of the underlying grey-level profile, which indicates the occurrence of a roof edge. Unfortunately, this scheme requires three thresholds to be given heuristically. Furthermore, it is an ill-posed task due to the embedded differentiation process (see Section 3.2).

The Haralick schemes of step [39] and roof [38] edge detection employ the process of least-square optimisation. A set of Chebychev orthogonal polynomials is used in these schemes to reconstruct the underlying grey-level surface of the image. Step edges and roof edges are then determined according to the corresponding criteria. Least-square optimisation methods minimise a well-posed quadratic energy function, where the solution is obtained by linear algebra [90]. Note that a regularisation formula can be transformed into a quadratic equation (see Section 3.3).

Recently, Chen and Yang proposed a step edge detector based on the Regularised Cubic B-Spline (RCBS) fitting [17]. However, this scheme has two limitations which degrade its performance. First, the Prewitt edge detector, an ill-posed operator, is used to indicate the local gradient, along which the subsequent fitting is applied. The accuracy of this scheme is thus limited. Second, the grey-levels along the gradient, which are required for the fitting, do not coincide with the grid pixels of the image (Figure 3.2(a)). Interpolation is thus required, which increases the computation time. This chapter presents a roof edge detector, derived from Chen and Yang's step edge detector, which overcomes the above two limitations. The proposed roof edge detector employs the 1-D RCBS fitting on the horizontal and the vertical orientations of a window of image pixels to generate two 1-D signals (see Figure 3.2(b)), which provides sufficient information of the 2-D facet to enable edge detection.



(a) (b)

Figure 3.2: (a) The mask of the Chen/Yang edge detector, where the orientation of the fitting is determined by the Prewitt operator. (b) The mask of the proposed roof edge detector.

3.2 Well-posedness and Regularisation

Hadamard defined a well-posed task to have the properties of existence, uniqueness and continuity [6], where

existence: for each datum g in a given class of functions G, there exists a solution x in a prescribed class X;

uniqueness: the solution x is unique in X;

continuity: when the error on the data g tends to zero, the induced error on the solution x also tends to zero.

As indicated in Chapter 1, the tasks of computer vision are inverse processes in the sense that the 3-D physical scene is reconstructed from digital images. The information contained in an image is insufficient to produce a unique and stable solution which represents the physical scene, unless an adequate physical knowledge is used to constrain the solution space. The incorporation of constraints is referred to as the regularisation process. For example, due to the process of differentiation in edge detection, a small perturbation of g (e.g. noise) induces an unpredictable change of the edge location x. Thus, it is an ill-posed process which violates the criterion of continuity. From the point of signal processing, the process of differentiation corresponds to a high-pass filtering, which enhances the noisy components of the signal. Hence, the task is ill-posed and the solution of x is numerically unstable.

Tikhonov's theory of regularisation provides a method to convert an ill-posed process into a well-posed process [93]. The principle of Tikhonov's regularisation is to employ adequate constraints in the process [6, 92]. A typical ill-posed task, which corresponds to image restoration or the curve/surface fitting in edge detection, is to find a function $f(\cdot)$

from the data $g(\cdot)$ such that $f(\cdot) \approx g(\cdot)$. Tikhonov's format includes an additional term which comprises a suitable norm $\| \cdot \|$ as well as a stabilising function Q, i.e.

$$E = \| f(\cdot) - g(\cdot) \|^2 + \alpha \| Qf(\cdot) \|^2, \tag{3.1}$$

where α determines the degree of regularisation and E is the energy. Thus $f(\cdot)$ is determined such that E is minimised.

L_2 norm is generally chosen as $\| \cdot \|$ and d^2/dx^2 as Q for the sake of simplicity [94], i.e.

$$\| Qf(\cdot) \|^2 = \begin{cases} \int (\frac{d^2 f(x)}{dx^2})^2 dx, & (1-D) \\ \int\int (\nabla^2 f(x,y))^2 dx dy. & (2-D) \end{cases}$$

In this way, the space of the solution for a regularised functional is constrained by suppressing the second order derivative of $f(\cdot)$, i.e. a smoothness constraint. The physical justification is that the noise-free image is band-limited by the optics, therefore, all its derivatives of the underlying surface should exist and be bounded [94].

Toore and Poggio argued that the determination of $f(\cdot)$ via the Tikhonov regularisation is equivalent to convolving the digitised image $g(\cdot)$ with a a cubic spline filter, which is very similar in shape to a Gaussian filter [94]. On the other hand, the role of Gaussian smoothing can be replaced by a regularised fitting. This is the basic concept of Bounded Diffusion, which will be discussed in Chapter 4.

3.3 Regularised Cubic B-Spline Fitting

The Regularised Cubic B-Spline (RCBS) fitting, which employs the principle of the regularisation theory [6, 92], is used to reconstruct the grey-level profile $f(x)$ from a discrete array of data $g(x_j)$ in a well-posed way, i.e.

$$E = \sum_j (f(x_j) - g(x_j))^2 + \alpha \int (\frac{d^2 f(x)}{dx^2})^2 dx, \tag{3.2}$$

where α is a positive number and x_j are the sampled pixels of a spatial coordinate x. The variational approach is commonly used to solve the functionals [92]. However, techniques such as curve fitting can transform a functional problem into the minimisation of a quadratic energy equation, where the solution is easily determined by linear algebra. Among various curves available for the fitting, a spline under a certain premise has a least value of the second order derivative according to the Holladay theorem [1]:

Holladay Theorem

Let $\triangle : a = x_1 < x_2 < ..... < x_M = b$, *and a set of real numbers* $\{g(x_k)\}(k = 1, 2, ..., M)$ *be given. Then among all the functions f(x) with a continuous second derivative on [a,b], and such that* $f(x_k) = g(x_k)$, *the spline function* $S_\triangle(x)$ *with junction points at* x_k *and with* $S''_\triangle(a) = S''_\triangle(b) = 0$ *minimise the intergal*

$$\int_a^b (f''(x))^2 dx.$$

Here the spline function $S_\triangle(x)$ *is defined to be composed of cubic polynomials in each sub-interval* $x_{k-1} \leq x \leq x_k$ $(k = 2, 3, ..., M)$, *and satisfies* $S'_\triangle(x_k+) = S'_\triangle(x_k-)$ *as well as* $S''_\triangle(x_k+) = S''_\triangle(x_k-)$, $(k : 2, 3, ..., M-1)$.

Since the sampling rate of a digital image is fixed, the interval $[x_{k-1}, x_k]$ is equidistant. Hence a spline $S_\triangle(x_k)$ is a linear combination of a third-order basis function provided that the derivatives of the basis function are zero at the boundary points. This concept motivates the RCBS fitting [17, 56], which uses the cubic B-spline Ω (Figure 3.3) as the basic elements of the fitting:

$$\Omega = \begin{cases} 0 & 2 \leq |x| \\ -|x|^3/6 + |x|^2 - 2|x| + 4/3 & 1 < |x| < 2 \\ |x|^3/2 - |x|^2 + 2/3 & |x| \leq 1 \end{cases}$$

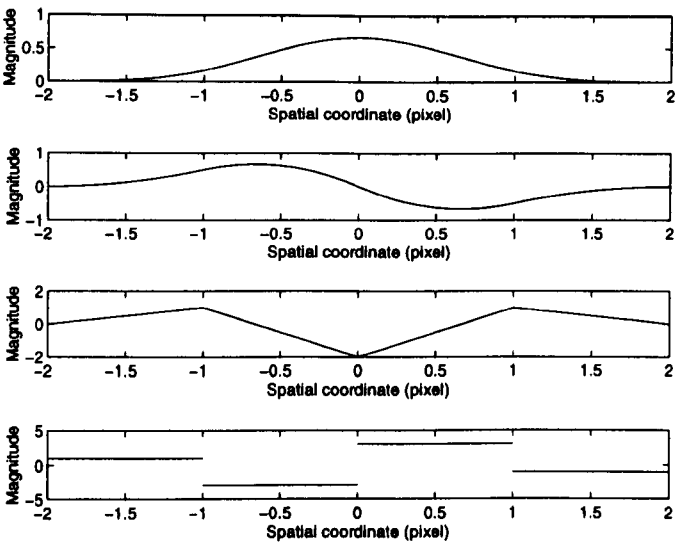


Figure 3.3: From top to bottom: cubic B-spline and its first, second and third order derivatives.

By shifting the basis function over the interval [1,M],

$$e_i(x) = \Omega(x - i) \qquad 0 \leq i \leq M + 1 \qquad i \in \mathbf{Z}.$$

The solution $f(x)$ in equation (3.2) is then represented as the linear combination of the basis functions:

$$f(x) = \sum_{i=0}^{M+1} c_i e_i(x), \tag{3.3}$$

where c_i is the coefficient of the basis function e_i.

In this way, the regularised functional in equation (3.2) becomes a quadratic energy equation:

$$E = \sum_{j=1}^{M} (\sum_{i=0}^{M+1} c_i e_i(x_j) - g(x_j))^2 + \alpha \sum_{j=1}^{M} (\sum_{i=0}^{M+1} c_i \frac{d^2 e_i(x_j)}{dx^2})^2, \tag{3.4}$$

The first term of the above equation is the summation of the square of the differences between $f(x_j)$ and $g(x_j)$, while the second term determines the summation of the second order derivatives in x_j. Minimising E represents the reduction in the differences as well

as the second order derivatives at the positions of x_j, and α is used to adjust the relative importance between the two terms. Given $g(x_j)$ and α, c_i is then determined. The solution $f(x)$ and its derivatives can then be easily obtained by multiplying c_i with the corresponding derivatives of e_i:

$$f^{(n)}(x) = \sum_{i=0}^{M+1} c_i e_i^{(n)}(x),$$

where $f^{(n)}(x)$ and $e_i^{(n)}(x)$ represent the nth order derivatives of $f(x)$ and $e_i(x)$, respectively.

3.3.1 Quadratic Energy Equation

Define

$$\mathbf{C} \equiv \begin{bmatrix} c_1 \\ \vdots \\ c_{M+1} \end{bmatrix} \qquad \mathbf{A} \equiv \begin{bmatrix} e_1(x_o) & \cdots & e_M(x_o) \\ \vdots & & \vdots \\ e_1(x_{M+1}) & \cdots & e_M(x_{M+1}) \end{bmatrix}$$

$$\mathbf{A_2} \equiv \begin{bmatrix} \frac{d^2 e_1(x_0)}{dx^2} & \cdots & \frac{d^2 e_M(x_0)}{dx^2} \\ \vdots & & \vdots \\ \frac{d^2 e_1(x_{M+1})}{dx^2} & \cdots & \frac{d^2 e_M(x_{M+1})}{dx^2} \end{bmatrix} \qquad \mathbf{G} \equiv \begin{bmatrix} g(x_1) \\ \vdots \\ g(x_M) \end{bmatrix},$$

where $\mathbf{A}$, $\mathbf{A_2}$ and $\mathbf{G}$ are known, and $\mathbf{C}$ is the unknown variable. In this way, equation (3.4) can be represented as

$$\begin{aligned} \mathbf{E} &= (\mathbf{A^T C} - \mathbf{G})^{\mathbf{T}}(\mathbf{A^T C} - \mathbf{G}) + \alpha(\mathbf{A_2^T C})^{\mathbf{T}}(\mathbf{A_2^T C}) \\ &= (\mathbf{A^T C})^{\mathbf{T}}(\mathbf{A^T C}) - \mathbf{G^T A^T C} - (\mathbf{A^T C})^{\mathbf{T}}\mathbf{G} + \mathbf{G^T G} + \alpha \mathbf{C^T A_2 A_2^T C} \\ &= \mathbf{C^T}(\mathbf{A A^T} + \alpha \mathbf{A_2 A_2^T})\mathbf{C} - 2\mathbf{C^T A G} + \mathbf{G^T G}. \end{aligned}$$

Define

$$\mathbf{P} = \mathbf{A A^T} + \alpha \mathbf{A_2 A_2^T}$$

$$= \begin{bmatrix} \mathbf{A} & \mathbf{A_2} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \alpha\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A^T} \\ \mathbf{A_2^T} \end{bmatrix}.$$

Since the row vectors of $[\mathbf{A} \quad \mathbf{A_2}]$ are independent of each other, the matrix $\mathbf{P}$ is positive definite as long as $\alpha > 0$. Therefore,

$$\mathbf{E} = \mathbf{C^T P C} - 2\mathbf{C^T A G} + \mathbf{G^T G}.$$

This is a standard quadratic energy equation and the graph is a paraboloid in the (M+2) dimensional hyperspace. The minimum of $\mathbf{E}$ occurs at $\mathbf{C} = \mathbf{P^{-1}AG}$ [90].

3.3.2 Ill-Conditioness

A quadratic energy equation is well-posed. However, it can be ill-conditioned, which means that a small perturbation in the input signals results in a large variation of the output. When $\alpha \rightarrow 0$, $\mathbf{P} \rightarrow \mathbf{AA^T}$, which is a singular matrix because the rank of $\mathbf{A}$ is M, and the dimension of the matrix $\mathbf{AA^T}$ is M+2 by M+2 such that $det(\mathbf{AA^T}) = 0$. The singularity of $\mathbf{AA^T}$ causes the computation of $\mathbf{C}$, which requires the inverse of $\mathbf{P}$, to be ill-conditioned. Since there is no clear boundary between ill-conditions and well-conditions [6], the MATLAB software is used to simulate the fitting with a decreasing value of α. The results (Figure 3.4) show that when α equals to $10^{-7}, 10^{-10}, 10^{-13}$, satisfactory fittings are achieved. However, when α is further decreased to $10^{-16}, 10^{-19}$ as well as 0, the results of the fitting are unsatisfactory, i.e. the results do not represent roof edges. Similarly, $\alpha \rightarrow \infty$ causes the computation to be ill-conditioned. Values of α in the range 10^{-7} to 20 showed similar results to Figure 3.4(a).

In the RCBS fitting, the number of coefficients (M+2) exceeds the number of grid pixels (M) to be fitted. As a result Chen and Yang [17] proposed the exact mapping (EM)

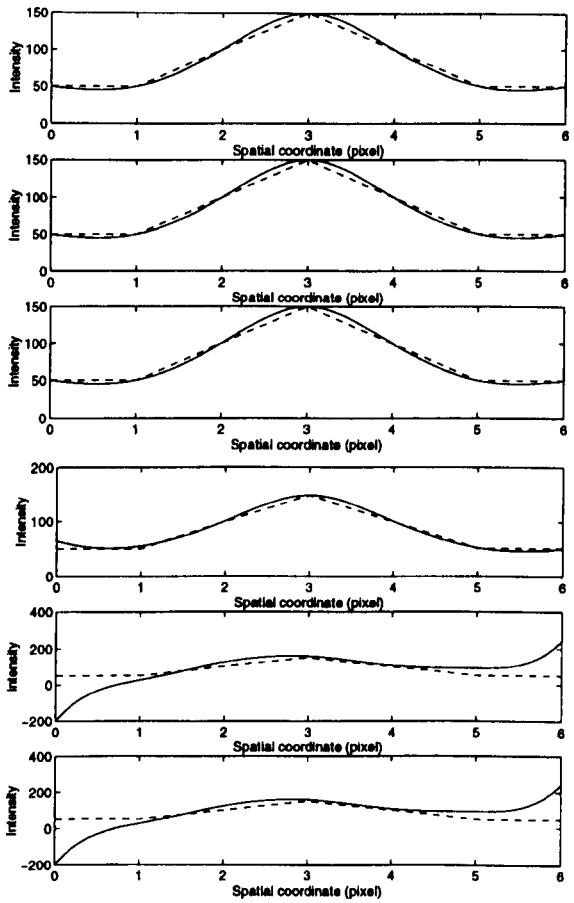
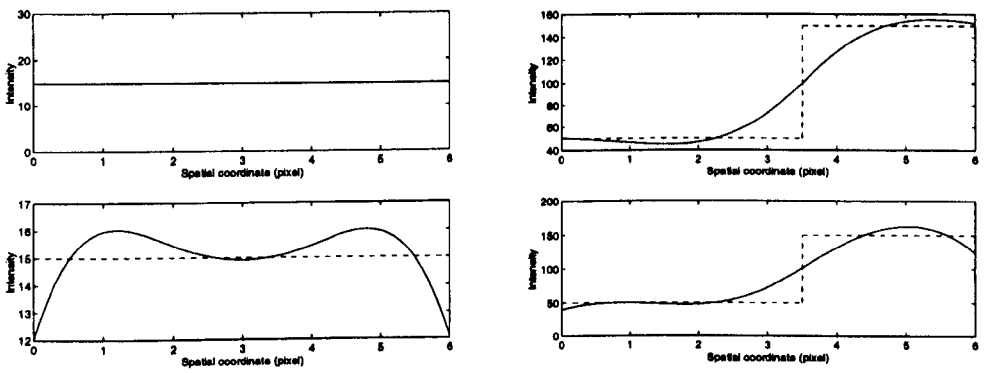


Figure 3.4: The fitted curve (solid line) on a roof edge (dashed line) using the RCBS fitting with various regularisation factors α. From top to bottom: α=10^{-7}, 10^{-10}, 10^{-13}, 10^{-16}, 10^{-19}, and 0.

method which changes equation (3.3) to:

$$f(x) = \sum_{i=1}^{M} c_i e_i(x)$$

Therefore, only M cubic B-splines are used instead of M+2 as in equation (3). However, the use of EM will not result in an optimum fit, because the RCBS fitting is regularised and therefore, is well-posed. The solution of a well-posed problem is guaranteed to exist, is unique, and is continuously dependent on the input data (see Section 3.2). If the boundary cubic B-splines (i.e. $e_0(x)$ and $e_{M+1}(x)$) is removed from the fitting procedure as suggested in [17], then the accuracy of the fitted curve degrades as illustrated in Figure 3.5. In this thesis the RCBS fitting with M+2 cubic B-splines is used.



(a) (b)

Figure 3.5: Top of (a): the fitted curve (solid line) of a straight line (dashed line) using the RCBS fitting; Top of (b): the fitted curve (solid line) of a step edge (dashed line) using the RCBS fitting; Bottom of (a) and (b): the fitted curve obtained by the exact mapping (EM) method. Here α=0.1.

3.3.3 Theorem of Linear Fitting

The regularised fitting is a linear process in the sense that the fitted curve is proportional to the sampled value according to the following linear fitting theorem:

Theorem 1 Linear Fitting Theorem

Consider the regularised fitting

$$E(f,g) = \int (f-g)^2 dx + \alpha \int (\frac{d^2 f}{dx^2})^2 dx,$$

the solution f of the functional E(f,g) is linearly dependent on the image grey-level g.

Proof :

Given g_o,

Let f_o *be the candidate solution of* g_o *which minimises E, i.e.* $E(f_o, g_o)$ *is minimised.*

then

for $g_m = a \times g_o + b$, ($a,b \in \mathbf{R}$)

there exists $f_m = a \times f_o + b$, *such that*

$$E(f_m, g_m) = \int (f_m - g_m)^2 dx + \alpha \int (\frac{d^2 f_m}{dx^2})^2 dx,$$

$$= \int [a(f_o - g_o)]^2 dx + \alpha \int (\frac{d^2 (a f_o)}{dx^2})^2 dx,$$

$$= a^2 E(f_o, g_o),$$

is also minimised.

3.4 The Design of a Roof Edge Detector

A roof edge is defined as a discontinuity in the first order derivative of a 1-D grey-level profile g [54], which is fitted by a continuous function f. Thus, the location of a roof edge corresponds to the extremum in the second order derivative f'', and the zero-crossing in the third order derivative f'''. The zero-crossings of f''' can accurately indicate the position of edges, but they are very sensitive to high-frequency signals such as noise. Thus, a threshold in f'' is introduced to make the criteria more robust against noise. As a result the criteria for a roof edge are:

- $f'' \geq$ threshold;

- zero-crossing occurs in f'''.

Since an image is a 2-D signal, the above 1-D criteria cannot be used directly. There are two approaches to solve this problem: (1) augment the 1-D criteria to 2-D; and (2) use a 1-D signal in an appropriate orientation to represent the 2-D image. The latter approach is used in this design to simplify the computation.

3.4.1 Principal Cross Section

The grey level of a 2-D image describes a 3-D surface. To use a 1-D signal f (a grey-level profile) to represent this 3-D surface for roof edge detection, a cross-sectional plane is required on which f is the projection of the 3-D surface. This cross-sectional plane, referred to as the principal cross section, is perpendicular both to the 2-D image and the isophote curves (i.e. the curves which connect pixels of the same grey level). The orientation of the principal cross section is thus referred to as the principal orientation (see Figure 3.1).

Let $S(x, y)$ be the 3-D grey-level surface, where x and y are spatial coordinates. Let $\hat{x}$ be the principal orientation, and $\hat{y}$ be the orientation of the isophote curve, then $\frac{\partial S}{\partial \hat{y}} = \frac{\partial^2 S}{\partial \hat{y}^2} = 0$. Given an arbitrary direction t inclined at an angle θ to the direction of $\hat{x}$, then

$$\hat{x} = t cos\theta \qquad \text{and} \qquad \hat{y} = t sin\theta.$$

Therefore,

$$\begin{aligned} \frac{dS}{dt} &= \frac{\partial S}{\partial \hat{x}} \times cos\theta + \frac{\partial S}{\partial \hat{y}} \times sin\theta \\ &= \frac{\partial S}{\partial \hat{x}} \times cos\theta, \end{aligned} \tag{3.5}$$

and

$$\begin{aligned}\frac{d^2S}{dt^2} &= \frac{\partial^2 S}{\partial \hat{x}^2} \times cos^2\theta + 2 \times \frac{\partial^2 S}{\partial \hat{x}\partial \hat{y}} \times sin\theta \times cos\theta + \frac{\partial^2 S}{\partial \hat{y}^2} \times sin^2\theta \\ &= \frac{\partial^2 S}{\partial \hat{x}^2} \times cos^2\theta. \end{aligned} \tag{3.6}$$

Denote

$$f'_P = \frac{\partial S}{\partial \hat{x}}, \qquad f''_P = \frac{\partial^2 S}{\partial \hat{x}^2},$$

and

$$f'_\theta = \frac{dS}{dt}, \qquad f''_\theta = \frac{d^2S}{dt^2}.$$

Therefore, equations (3.5) and (3.6) become

$$f'_\theta(x,y) = f'_P(x,y) \times cos\theta,$$

$$f''_\theta(x,y) = f''_P(x,y) \times cos^2\theta. \tag{3.7}$$

These relationships infer that f along the principal orientation ($\theta = 0$) has the maximum first and second order derivatives. Note that these relationships are obtained under the assumption that the isophote curves are parallel to the roof edges. Although this is not always true, the assumption generally approximates the real cases.

3.4.2 Horizontal-Vertical Decomposition

To obtain the derivatives along the principal orientation (i.e. $f'_P(x,y)$ and $f''_P(x,y)$), a 2-D image is decomposed into two 1-D signals which are perpendicular to each other, e.g. the horizontal and the vertical components of the signal. The derivatives along the principal orientation are then determined from the derivatives of these signals. Given a cross section with an angle ($\theta + 90^o$), equation (3.7) becomes

$$f'_{\theta+90^o}(x,y) = f'_P(x,y) \times cos(\theta + 90^o) = f'_P(x,y) \times sin\theta,$$

$$f''_{\theta+90^o}(x,y) = f''_P(x,y) \times cos^2(\theta + 90^o) = f''_P(x,y) \times sin^2\theta.$$

Therefore,

$$\begin{aligned}(f'_\theta(x,y))^2 + (f'_{\theta+90^o}(x,y))^2 &= (f'_P(x,y))^2 cos^2\theta + (f'_P(x,y))^2 sin^2\theta \\ &= (f'_P(x,y))^2, \end{aligned} \tag{3.8}$$

$$f''_\theta(x,y) + f''_{\theta+90^o}(x,y) = f''_P(x,y)cos^2\theta + f''_P(x,y)sin^2\theta = f''_P(x,y). \tag{3.9}$$

Equation (3.8) and (3.9) show that $f'_P(x,y)$ and $f''_P(x,y)$ can be obtained from the derivatives along any two perpendicular orientations. Note the two formulae in equation (3.8) and (3.9) are identical to the definition of the gradient and the Laplacian respectively, where $f'_P(x,y)$ corresponds to the gradient, and $f''_P(x,y)$ corresponds to the Laplacian. To simplify the computation, the two perpendicular orientations are chosen to be the horizontal and the vertical orientations so that the grey levels on the sampled image can be directly used for the regularised fitting.

3.4.3 The Algorithm

In the proposed roof edge detector, the RCBS fitting is applied along the horizontal and the vertical orientations to reconstruct two continuous grey-level profiles (Figure 3.2(b)). The derivatives of the fitted curve f along these two orientations are then obtained as described in Section 3.3. Since the basis function, the cubic B-spline, is a third order piecewise polynomial, the third order derivative of the fitted curve f is not continuous at the grid pixels (see Figure 3.3).

The criteria for detecting a roof edge (Section 3.4) are then applied along the principal orientation. The first criterion requires f''_P, determined using equation (3.9), to be greater than a threshold. The second criterion requires a zero-crossing in f''', i.e. $f'''_P(t_o+) \times f'''_P(t_o-) < 0$. To simplify the application of these criteria, the third order derivatives are

examined along the horizontal and the vertical orientations. If a sign change occurs along either of the two orientations, the second criterion is satisfied. The first criterion is then used to detect the roof edge.

The magnitude of f''_P is used as an indication of roof edges in the proposed scheme, where f''_P is determined from two regularised 1-D signals, and equals approximately to the Laplacian of Gaussian $\nabla^2 G(\sigma, x, y) * S$. The difference between f''_P and $\nabla^2 G(\sigma, x, y) * S$ is the shape of the kernel, i.e. a cross kernel (Figure 3.2(b)) is used to compute f''_P, while a square kernel is used to compute $\nabla^2 G(\sigma, x, y) * S$.

The pseudo codes of the proposed roof edge detector are:

begin

Input (Image(x, y), α, threshold);

Assign n = the size (in pixels) of the image;

For $(x = 1, 2, ...n)(y = 1, 2, ...n)$ **do**

begin

Apply the RCBS fitting along the horizontal orientation;

Determine $f''_{horizontal}(x, y)$, $f'''_{horizontal}(x-, y)$, and $f'''_{horizontal}(x+, y)$;

Apply RCBS fitting along the vertical orientation;

Determine $f''_{vertical}(x, y)$, $f'''_{vertical}(x, y-)$, and $f'''_{vertical}(x, y+)$;

If $(f'''_{horizontal}(x-, y) \times f'''_{horizontal}(x+, y) < 0)$

or $(f'''_{vertical}(x, y-) \times f'''_{vertical}(x, y+) < 0)$

then

begin

$f''_P(x, y) = |f''_{horizontal}(x, y) + f''_{vertical}(x, y)|$

If $f''_P(x, y) > threshold$

then Set EdgeMap(x, y)=Edge;

else Set EdgeMap(x, y)=Not an Edge;

end

end

Output (EdgeMap(x, y));

end

3.5 Performance Evaluations

The values of the regularisation factor α and the threshold are required for the proposed roof edge detector. The value of α determines the degree of smoothing, therefore, a large value of α should be used when the image is noisy. Conversely, if the image is noise-free, a small value of α should be used so as to preserve the image details. The value of the threshold reflects the size of an edge, which is defined as the difference of the slopes at the two sides of the roof edge [54]. The larger the edges to be detected, the larger the threshold.

Several synthetic images are used to test the proposed roof edge detector. These synthetic images contain a roof edge of a "ring" shape so that the performance of the edge detector in all orientations can be measured. The advantage of using synthetic images is that the true edge positions are known and therefore, quantitative evaluations are possible. The measurements of performance is the False-Correct Ratio (FCR) [17], which is determined by first classifying the pixels in the edge map to be true-positive, true-approximate, true-missing, neutral-extra, and false-positive; and regarding the number of falsely-detected pixels as the sum of true-missing (ntm) and false-positive pixels (nfp), and the number of correctly-detected pixels as the sum of the true-positive (ntp), true-approximate (nta), and neutral-extra (nne) pixels. FCR is the ratio of the number of

falsely-detected pixels to the number of correctly-detected pixels, i.e.

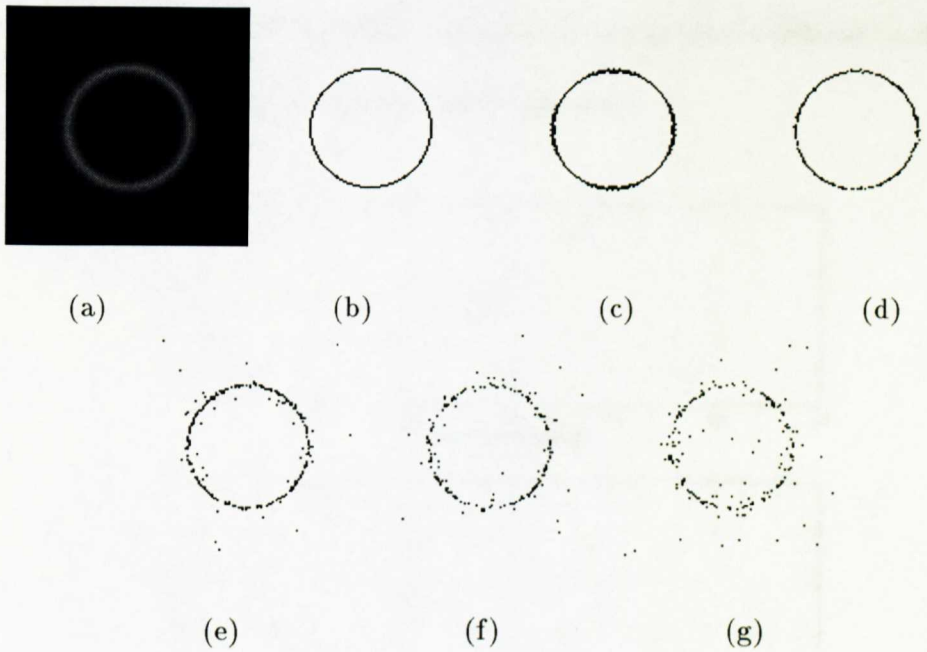
$$FCR = \frac{\text{number of falsely-detected pixels}}{\text{number of correctly-detected pixels}}$$

$$= \frac{\text{ntm+nfp}}{\text{ntp+nta+nne}}$$

The lower the value of FCR, the better is the performance of the operator. If the value of FCR is greater than 1, the operator is considered to have failed, because the number of falsely-detected pixels exceeds the number of correctly-detected pixels. Here the parameters which generate an edge map with the least value of FCR are considered optimum parameters.

In test 1, a synthetic image which contains a roof edge with a slope of 20 at both sides of the edge, i.e. the size of the edge is 40, is used to represent a typical roof edge (Figure 3.6(a)). The image is contaminated with zero-mean Gaussian noise of various standard deviation (NSD), and the FCR's of the edge maps produced by the proposed roof edge detector are determined. In the noise-free case, the edge map has a FCR of 0 (i.e. the edge map is perfect in the context of FCR) when the regularisation factor is 0.1 and the threshold is 10. The threshold is then set to 10 for all the noise levels since the regularisation factor alone should reflect the noise level. The optimum regularisation factors and their corresponding FCR's are then measured under various NSD's. The results (Figure 3.7) show that the optimum regularisation factor increases as the NSD increases. The corresponding optimum edge maps are shown in Figure 3.6.

In test 2, noise-free images are used and the size of the roof edge is decreased gradually so as to determine the effect of the threshold on the performance of the edge detector. The result is shown in Figure 3.8. As the size of the roof edge decreases, the optimum thresholds become smaller. The perfect edge map (i.e. FCR=0) cannot be produced when the size is reduced to 10. This is because as the size is reduced, the edge and non-edge



(a) (b) (c) (d)

(e) (f) (g)

Figure 3.6: The optimum edge maps produced when the test image (edge size=40) is contaminated with noise of various standard deviations (NSD's). (a) The test image. (b) The ideal edge map. (c) NSD=0. (d) NSD=5. (e) NSD=10. (f) NSD=15. (g) NSD=20.

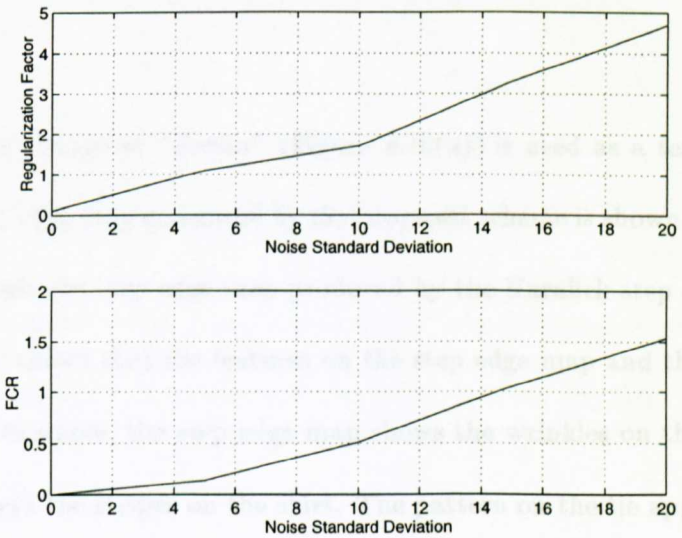


Figure 3.7: Top: The optimum regularisation factors obtained when the test image is contaminated with noise of various standard deviations (NSD's). Bottom: the corresponding FCR's when the optimum regularisation factors are used. In this test, the size of the roof edge and the threshold are 40 and 10 respectively.

points become more and more similar, and hence more and more difficult to distinguish. Figure 3.9 shows the corresponding optimum edge maps.

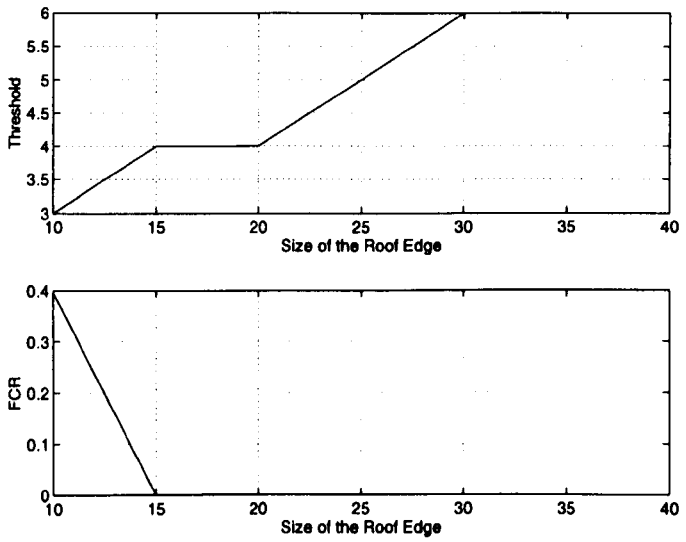


Figure 3.8: Top: The optimum thresholds obtained in noise-free images with roof edges of various sizes. Bottom: The corresponding FCR's. In this test, the regularisation factor α is 0.1.

In test 3, a real image of "Trevor" (Figure 3.10(a)) is used as a test image, and the corresponding roof edge map generated by the proposed scheme is shown in Figure 3.10(b). Figure 3.10(c) shows the step edge map produced by the Haralick step edge detector [39] for comparison. It shows that the features on the step edge map and the roof edge maps are different. For example, the step edge map shows the wrinkles on the shirt, while the roof edge map shows the stripes on the shirt. The pattern on the tie appears to be better detected using the proposed detector than the Haralick step edge detector.

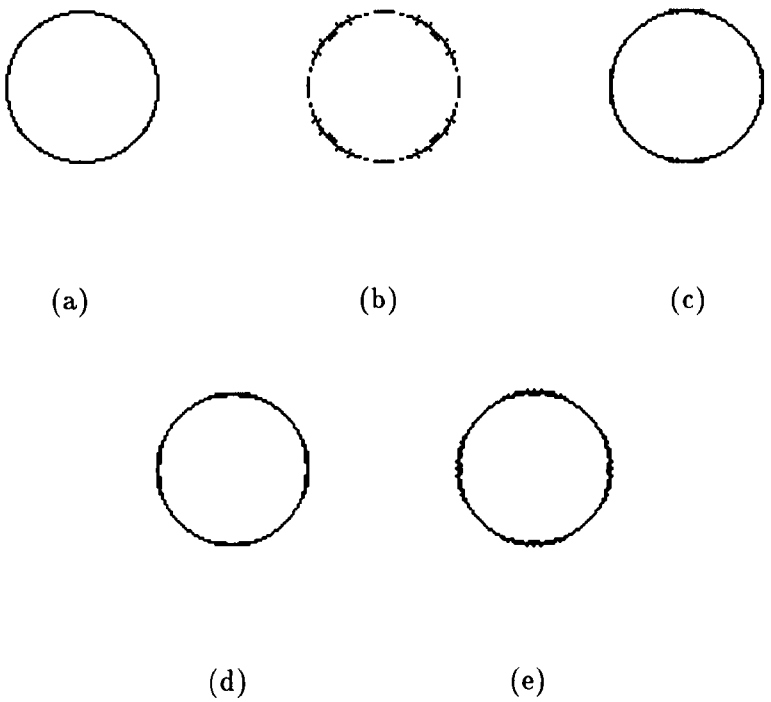


Figure 3.9: The optimum edge maps obtained for noise-free roof edges of various sizes: (a) The ideal edge map; (b) Edge size=10; (c) Edge size=20; (d) Edge size=30; (e) Edge size=40.

(a)

(b)

(c)

Figure 3.10: (a) The "Trevor" image. (b) The edge map of "Trevor" produced by the proposed roof edge detector (α=0.1; threshold=6). (c) The edge map of "Trevor" produced by the Haralick step edge detection scheme with the threshold of 4.

3.6 Summary

In this chapter, the concept of regularisation is examined. This concept relates to both the numerical stability (i.e. the well-posedness of the task) and the scale (Section 3.2). Section 3.3.1 shows the use of Cubic B-spline fitting transforms a functional of regularisation into a quadratic energy function. A roof edge detector is devised which does not rely on the Prewitt edge detector as in the Chen/Yang step edge detector [17]. The roof edge detector is much simpler than the Chen/Yang step edge detector, because the proposed Horizontal-Vertical decomposition enables a 2-D image to be analysed on a 1-D basis. The Regularised Cubic B-Spline fitting is also modified to achieve a better fitting (Section 3.3.2). Although the RCBS fitting is well-posed, the regularisation factor α should be carefully chosen to make the computation well-conditioned (Section 3.3.2).

Chapter 4

Bounded Diffusion

4.1 Multiscale Edge Detection

The multiscale aspect of edge detection was first examined by Rosenfeld and Thurston [85]. They analysed the edge responses using the box-shaped kernels with various sizes, and observed that some edge points are not detected when a large kernel (i.e. low spatial resolution) is applied. Marr and Hildreth proposed the Laplacian of Gaussian edge detector, in which a Gaussian pre-filter is applied to regularise the ill-posed task of edge detection and to suppress noise (see Section 2.2). They observed that when an image is convolved with a Gaussian kernel of various standard deviation σ, different edge maps, defined as the zero-crossings of the Laplacian, are produced. A small σ results in an edge map with more details than one which is obtained using a large σ, but there are also more noise-induced responses [66]. Marr and Hildreth suggested that the zero-crossings which exist over several scales should be considered as physically significant [66]. These edges are referred to as salient edges.

Witkin proposed the scale-space filtering, which shows the evolutionary behaviour of an edge under different scales [108]. A scale space is spanned by the spatial coordinate and

a scale coordinate, where the scale is the standard deviation σ of the Gaussian pre-filter. In a scale space, the zero-crossings of the second order derivative of a signal produce traces which reflect the relationship between the scale and the step edges. Witkin pointed out the "well-behavedness" of the σ scale space, i.e. when the scale is varied from coarse to fine, new local extrema are created while existing ones remain [108]. Babaud *et al.* [3] proved that for a 1-D signal, this property only holds when the signal is convolved with a 1-D Gaussian. Yuille and Poggio [109] further confirmed that this property also holds when an image is convolved with a 2-D Gaussian, and proved that it can be applied to all level-crossing contours. Koenderink [53] proved that the Gaussian is the Green's function of the diffusion equation, i.e. in the 1-D case

$$\frac{\partial^2 \psi}{\partial x^2} = \frac{\partial \psi}{\partial \sigma}$$

where σ represents the scale, x is the spatial coordinate and ψ is the signal. Hence, the theory of scale space motivates the study on the diffusive aspects of an image.

Based on the evolutionary behaviour in the scale space, a few multiscale edge detection schemes have been proposed. The essence of these schemes is to locate edges as accurately as possible while suppressing noise. Bergholm proposed the Edge Focusing scheme [5] which first obtains an edge map from a coarse scale, and then consecutively decreases the scale to recover the true positions of edges. Lu and Jain [63] proposed the scheme for "reasoning about edges in scale space" (RESS), which involves a large amount of decision making to classify edges according to their behaviour in the scale space.

The scale defined in the Edge Focusing and RESS schemes are basically the standard deviation (σ) of the Gaussian pre-filter. Since σ relates to the shape and the range of the filter's impulse response, the value of the scale affects not only the smoothing effect on the signal, but also the kernel size. When a large σ is required to suppress noise, the resultant

kernel size is large. However, an edge is a local feature. A large kernel includes irrelevant information into the edge detection and therefore, the detected locations of edges deviate from their true positions. This is why in both Bergholm's, and Lu and Jain's approaches, extra computations are required to recover the true positions of edges. The α scale space is proposed in Section 4.2 to cope with this problem. In the α scale space, the size of the operator kernel is independent of the value of α, which corresponds to the degree of smoothing.

The value of a scale corresponds to the noise level of the image [16]. The higher the noise level, the stronger the smoothing effect should be. Hence, the noise level determines the lower bound of the scale, beyond which the estimated locations of edges are no longer accurate because the noise is not sufficiently suppressed. The Edge Focusing scheme employs a series of scale (σ = 4.2, 3.85, 3.5, 3.2, 2.8, 2.5, 2.1, 1.75, 1.4, 1.0, 0.7) which is heuristically chosen [5]. Therefore, if an image is very noisy, the edge locations recovered in the finest scale are not accurate. This is referred to as an "over-focused" phenomenon in [5]. In some images, the noise level varies from one region to another, thus the scale needs to be adjusted adaptively. This adaptivity is referred to as the "variable blurring" in [5]. To prevent over-focusing and to enable variable blurring in edge detection, a multiscale edge detector is proposed in Section 4.3 where the finest scale is adaptively adjusted according to the local noise level.

Bergholm claimed that noise and unnecessary edge details are both eliminated in the Edge Focusing scheme by the thresholding in the coarsest scale. In the successive stages (at finer scales), the Canny operator [16], but without the thresholding process, is used to detect edges [5]. However, in a very noisy image, some noise responses are not eliminated in the coarsest scale, and they develop into noise clusters (see Section 4.4 and Figure 4.14(c), 4.17(c)). This is because there is no thresholding process in the successive scales

to suppress noise. Note that the criterion of an edge being the zero-crossings of the second order derivative is susceptible to noise, thus it is normally accompanied by the criterion of a large gradient (see Sections 2.2 and 2.3). To prevent noise clusters in edge detection, a series of thresholds should be used in every scale in a multiscale scheme.

A multiscale edge detector with a fixed-size kernel is proposed to address the above three issues. The scale is adjusted adaptively according to the local noise level. A series of thresholds are used in every scale, which controls the amount of details to be preserved in the edge map. The proposed multiscale edge detector is still based upon the Regularised Cubic B-Spline (RCBS) fitting described in Section 3.3, where a set of cubic B-splines (Figure 3.3) is used to approximate the underlying 1-D grey-level profile. A regularisation term, controlled by a factor α, is introduced to suppress the effect of noise:

$$E = \sum_j (f(x_j) - g(x_j))^2 + \alpha \sum_j (\frac{d^2 f(x_j)}{dx^2})^2 dx, \tag{4.1}$$

where $g(x_j)$ denotes the grey level at the spatial coordinate x_j. The fitted curve $f(x)$ is determined by minimising the functional E. The proposed edge detector is thus referred to as the Multiscale edge detector based on Regularised Cubic B-Spline fitting (MRCBS).

In equation (4.1), α reflects the degree of smoothing, which is determined by the value of σ in Gaussian pre-filtering methods. Also, both the Gaussian pre-filtering and the regularised fitting convert the ill-posed nature of edge detection to well-posed [94]. As the kernel size is independent of α, the image is diffused within a small range, i.e. a bounded diffusion. Therefore α could be interpreted as a scale, where the terminology of scale is used in a more generalised manner, i.e. the scale is a parameter which controls the degree of smoothing.

4.2 Bounded Diffusion in α Scale Space

4.2.1 Uncertainty and Bounded Diffusion

A signal contains both global and local information, where a scale such as σ normally reflects the spatial extent of the operator kernel. The essence of a multiscale analysis is to interpret a signal using the operator kernel with varous spatial extents. As indicated in equation (1.3), the spatial resolution of Δx and the frequency resolution of Δw in a joint space-frequency (space-scale) analysis are constrained by the uncertainty principle [20]. This can be illustrated by two simple examples. The Fourier transform (FT) of a Dirac function ($\delta(x)$) is 1 for all frequency w. Similarily, the FT of a function that approximates an edge:

$$edge(x) = \begin{cases} 1 \ when \ x > 0, \\ -1 \ when \ x < 0; \end{cases}$$

is $2/jw$, where j is the unit imaginary number.

These two examples illustrate that the energy of a local feature in an image, e.g. a delta function or a step edge, spreads out over the entire frequency spectrum. This is also illustrated by the scale-space theory, which shows that an edge is scale (i.e. the frequency band) invariant [108]. On the other hand, although a single frequency function, e.g. $sin(x)$, is well localised in the frequency domain, it cannot be defined using a small number of pixels. Therefore, if an accurate frequency is required as in the analysis of textures, then the signal has to be analysed in a large interval in order to achieve a high resolution in the frequency domain. Conversely, if a feature has to be spatially localised,

then a small kernel with a fixed size is preferable in order to maintain the same spatial resolution.

Since an edge is defined as the union of the discontinuities (i.e. the local information) in the grey-level profile [54], the locations of the edge pixels are of primal concern. Therefore, a small and fixed-size operator kernel is preferred, which implies that the normal interpretation of a scale as σ is inadequate for edge detection. This is why the α scale space is proposed in Section 4.2.2 such that an image is diffused within a fixed-size kernel. A scale of α controls the degree of smoothing, but does not affect the size of the kernel. In this way, any irrelevant information is excluded from the smoothing process for noisy images.

4.2.2 α scale space

In this section, the diffusive/convergent behaviour of a signal which is regularised by the RCBS fitting is illustrated in the α scale space. The horizontal axis of this space is the spatial coordinate and the vertical axis is the scale in logarithm form. The logarithm form is used in order to visualise the evolutionary behaviour of an edge, depicted by dots, over a wide range of α. To generate the α scale space, three edge models: the isolated edge model, the pulse edge model, and the staircase edge model as in [110] are used (see Figure 4.1). An isolated edge model represents a single edge within an operator kernel. The pulse edge model and the staircase edge model are used to show the interaction between two edges which occur in an operator kernel. The size of the kernel is set to be 13, and the edge contrast in these models is set to 100.

The α scale space of the isolated edge model is shown in Figure 4.2(a). The central line is the true edge response, and the others are spurious responses, which are caused by the subtle oscillations of the fitted curve. Figure 4.2(a) shows that the edge response of

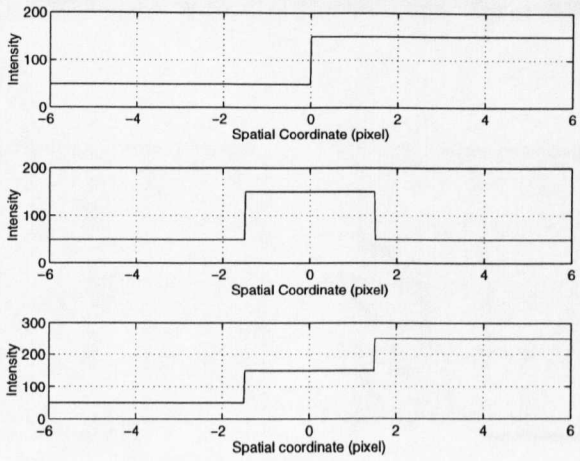
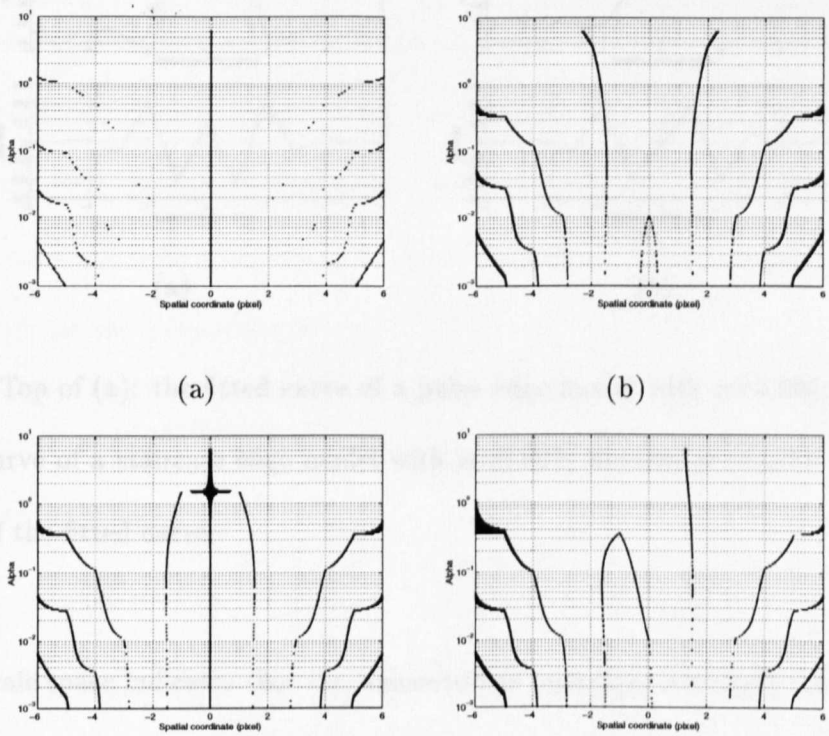


Figure 4.1: The edge models considered for the analysis in the α scale space. Top: the isolated edge model. Middle: the pulse edge model. Bottom: the staircase edge model.

an isolated edge does not deviate from its true position as α varies. This is similar to the σ scale space [62, 108]. Also, when the scale is varied from fine to coarse, the spurious responses drift away.

Figure 4.2(b) is the α scale space of the pulse edge model when the distance between the two edges is 3 pixels. It shows that the pair of true edge responses (which are between spatial coordinates ± 2 and which exist in all scales) converge to the true positions as $\alpha \rightarrow 0$, and they move away from each other as α increases. This phenomenon agrees with Witkin's localisation assumption [108], which suggests the use of the coarse-to-fine paradigm in the design of a multiscale scheme. As α increases, the spurious responses outside the true edges drift away while those inside merge with each other. Figure 4.2(c) is the α scale space of the staircase edge model. A phantom edge (the central line) exists between the true edges, and the true edges merge with the phantom edge at large value of α. When the two sides of the staircase edge have different contrast (left contrast of 50; right contrast of 100), the corresponding α scale space is shown in Figure 4.2(d). In

this case the phantom edge merges with the weaker edge. For a detailed discussion about phantom edges, see [19].



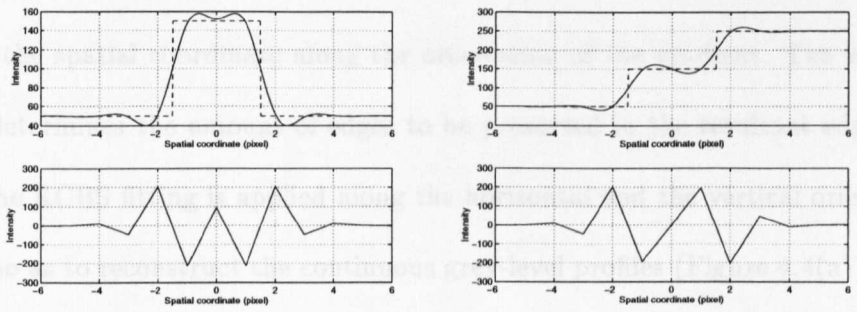
(a) (b)

(c) (d)

Figure 4.2: The α scale space of: (a) the isolated edge model; (b) the pulse edge model; (c) the staircase edge model; (d) the staircase edge model with a left contrast of 50 and a right contrast of 100.

Figure 4.3 shows that for the pulse edge model and the staircase edge model, the oscillations of the fitted curves result in zero-crossings in the second order derivatives of the fitted curves. These oscillations are subtle compared to the contrast of the edge. Note that the RCBS fitting is a linear process (see Section 3.3.3) and thus the amplitude of the oscillation is proportional to the contrast of the edge. Therefore, the spurious responses can be eliminated by the thresholding of the first order derivative of the fitted curve, which

is an essential process in edge detection (see Section 4.3 and [16, 94]).



(a) (b)

Figure 4.3: Top of (a): the fitted curve of a pulse edge model with α=0.001; Top of (b): the fitted curve of a staircase edge model with α=0.001; Bottom of (a),(b): The second derivative of the fitted curve.

The α scale space indicates that the coarse-to-fine paradigm is suitable for an α-based algorithm such that the detected edges will converge to their true positions. In addition, the deviations of edges from their true positions are caused by the interaction between edges which coexist in an operator kernel. Since the probability of two edges occurring in a small and fixed-size kernel is lower than in a varying-size kernel, α is more suitable than σ to be used as a scale in edge detection.

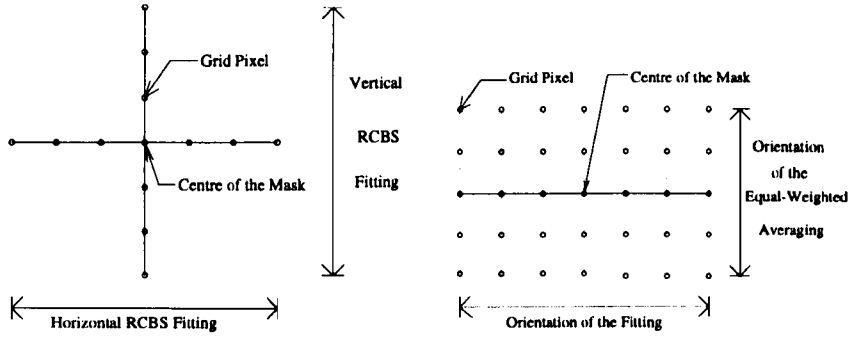
4.3 The Design of MRCBS

An edge is the union of discontinuities in the grey level of an image. In a regularised (smoothed) 2-D image f, an edge is defined as the union of pixels which correspond to the zero-crossings of the second order derivative along the orientation of the local gradient, and which is larger than some threshold [16, 94], i.e.

- Gradient > threshold;

- A zero-crossing of $\frac{\partial^2 f}{\partial n^2}$;

where n is the spatial coordinate along the orientation of the gradient. The value of the threshold determines the amount of edges to be preserved in the resultant edge map. In MRCBS, the RCBS fitting is applied along the horizontal and the vertical orientations of the image so as to reconstruct the continuous grey-level profiles (Figure 4.4(a)). The first and the second order derivatives along both orientations are measured (see Section 3.3) so as to determine the local gradient.



(a) (b)

Figure 4.4: (a) The operator kernel of MRCBS employed in every scale. (b) The operator kernel of MRCBS which is employed only in the finest scale. The orientation of the fitting is along the orientation of the gradient. The equal-weighted averaging is used to achieve anisotropic diffusion.

Based on the coarse-to-fine paradigm [5, 108], α is consecutively decreased in MRCBS until the lower bound of the scale is reached. The lower bound is determined by measuring the Energy of the High-Frequency component, which is defined in Section 4.3.1. For each scale, a threshold for the local gradient is adjusted in accordance with the scale, which is discussed in Section 4.3.2. In summary, the coarsest scale and a threshold are initially used. If the local gradient is greater than the threshold, and the scale is greater than the lower bound, a finer scale is then used so as to locate the edge with a better accuracy.

In the finest scale, the RCBS fitting is applied along the orientation of the gradient to measure $\frac{\partial^2 f}{\partial n^2}$. The zero-crossings of $\frac{\partial^2 f}{\partial n^2}$ are classified as edges. Figure 4.5 shows a schematic diagram of MRCBS.

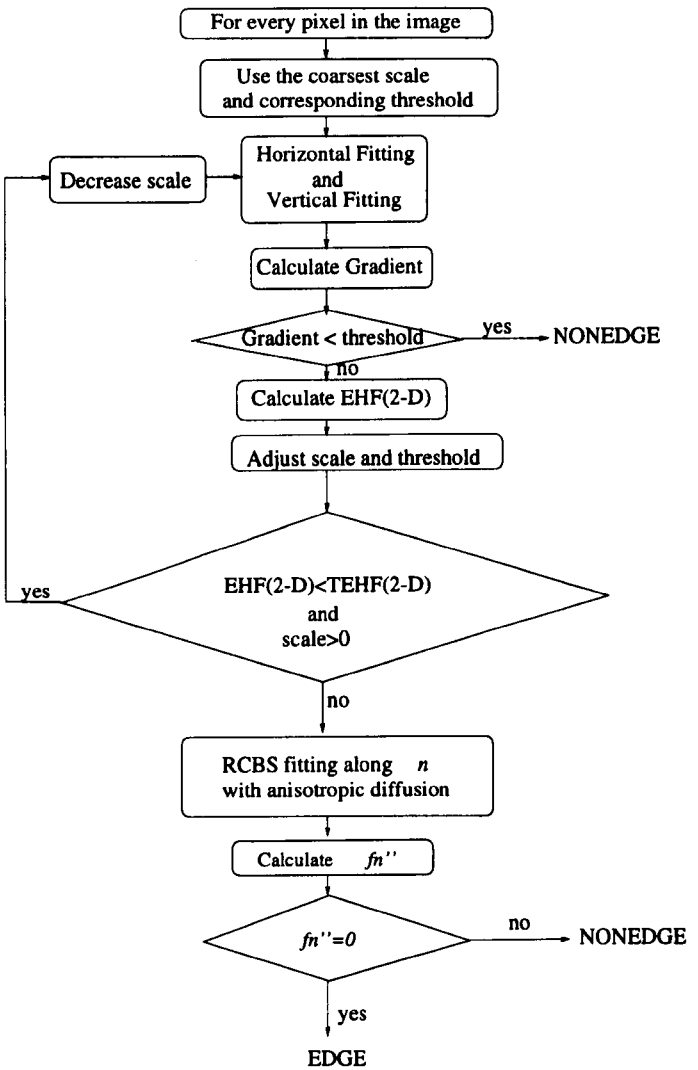


Figure 4.5: The schematic diagram of MRCBS.

4.3.1 Adaptivity in Scale

An index of the noise level is required to determine the lower bound of the scale. In the literature, several methods which adaptively adjust the regularisation factor α according to the image property have been proposed for image restoration [49, 68]. These methods

propose various models which induce α to converge iteratively to an appropriate value. All of these models assume α to be a monotonically increasing function of the noise level. However, different models result in different values of α for the same image.

It is commonly known that a low-pass filtering is suitable for suppressing noise because noise contains more high-frequency components than physical edges. For the same reason, a measurement of the high-frequency components of a signal indicates the local noise level. In the RCBS fitting, the following regularised functional is minimised:

$$E = \| f(\cdot) - g(\cdot) \|^2 + \alpha \| f''(\cdot) \|^2,$$

where $\| \cdot \|$ denotes the l^2 norm, $f(\cdot)$ denotes the fitted curve, and $g(\cdot)$ denotes the original image data. The first part of this functional $\| f(\cdot) - g(\cdot) \|^2$ is the residual energy of the fitting. The second part $\| f''(\cdot) \|^2$ represents the Energy of the High Frequency component (EHF), which provides an index of the noise level. To illustrate the relationship between noise level, α and EHF, a series of tests is conducted. The EHF's of a 1-D step edge with a contrast of 100 (the top plot of Figure 4.1) and contaminated by the Gaussian Noise of various Standard Deviations (NSD's) are measured. The RCBS fitting is used to regularise the noisy images. Each EHF is the average value of 10000 different noisy edges with the same noise level. The results (Figure 4.6) show that EHF is a function of NSD and α, with α inversely related to EHF. Also, a noisy edge has a higher EHF than a less noisy edge. Therefore, if the scale is consecutively decreased as in the coarse-to-fine paradigm, and a Threshold of EHF, referred to as TEHF, is used to terminate the decrement of the scale, then a noisy edge will cause the decrement of α to stop at a higher value. In this way, the value of α is adjusted automatically according to the local noise level, i.e. this is equivalent to variable blurring.

Applying TEHF is equivalent to cross-sectioning the surface in Figure 4.6 perpendicular

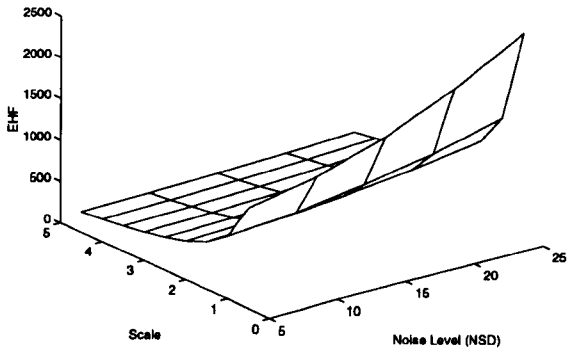


Figure 4.6: The relationship between the noise level (NSD), the scale of α and the EHF when the edge contrast is 100.

to the EHF-axis and in parallel to the NSD-scale plane. Although the scale (α) is a monotonically increasing function of the noise level (NSD), using a different value of TEHF results in a different relationship between them. To calibrate the NSD-α relationship by adjusting TEHF, an ideal step edge with a contrast of 100 (top plot of Figure 4.1), and contaminated by the Gaussian noise with NSD of 10, is assumed to be commonly encountered in real images. The RCBS fitting is applied to this signal with α ranging from 0.5 to 4.5, with an increment of 0.5. For each α, 10000 different noisy edges with NSD of 10 are fitted, and the edge is considered to be correctly localised if a zero-crossing of the second order derivative of the fitted curve occurs within ± 0.1 pixels of the edge. The number of correct localisations, a function of α, reaches its maximum when $\alpha = 3.5$ (Figure 4.7). To terminate the decrement of α at a value of 3.5 when NSD=10, a TEHF which is the average of the EHF's when $\alpha = 3.5$ and 4.0, and NSD=10, i.e. 326.05 and 289.99, is chosen. In this way, when $\alpha = 4.0$, the scale is decreased because EHF<TEHF; when $\alpha = 3.5$, the decrement of the scale is terminated because EHF>TEHF. Hence TEHF is set to be 308 and therefore, the NSD-α relationship as in Figure 4.8 is obtained,

in which $\alpha = 3.5$ when NSD= 10.

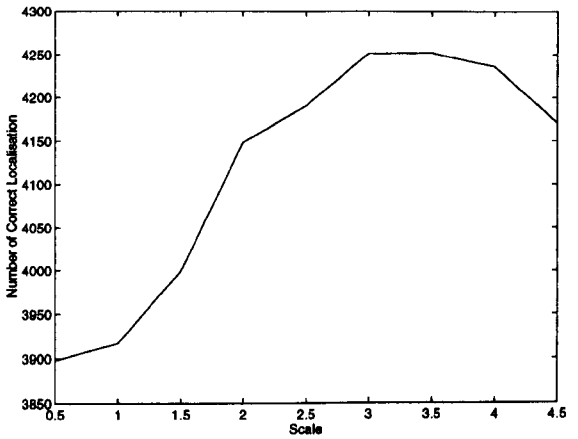


Figure 4.7: The numbers of correct localisations when the RCBS fitting at various scales is applied to 10000 different noisy edges. The contrast of the edge is 100 and the NSD is 10.

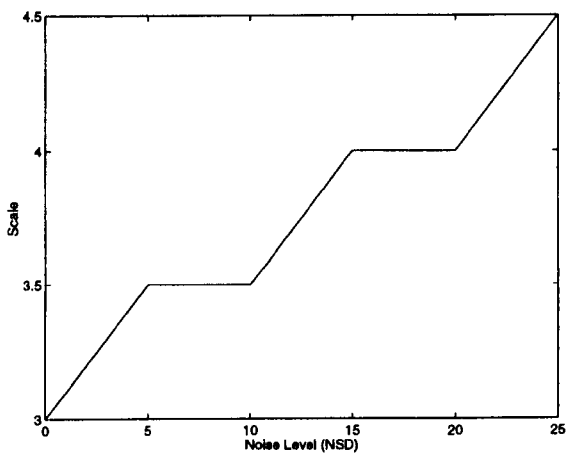


Figure 4.8: The relationship between α and NSD when TEHF is 308.

For a 2-D image, the RCBS fitting is applied along both the horizontal and the vertical orientations of the sampled image, thus the 2-D EHF is defined as follows:

$$EHF(\text{2-D}) \equiv \| f''_{horizontal}(\cdot) \|^2 + \| f''_{vertical}(\cdot) \|^2 .$$

Therefore, the decrement of the scale terminates when

$$EHF(\text{2-D}) > TEHF(\text{2-D}),$$

where the value of TEHF(2-D) is twice as large as that of TEHF, i.e. 616.

4.3.2 Scale-Threshold Consistency

As indicated in Section 4.1, a thresholding process in every scale is necessary for a multiscale scheme to prevent noise clusters. However, the slope of a regularised edge, the variable to be thresholded, varies with the different degree of smoothing (e.g. σ and α). Figure 4.9 shows an ideal step edge and the curves fitted by the RCBS fitting with various α: the higher the scale, the smaller is the slope. The thresholds, which distinguish the preserved and the eliminated edges, should have the same thresholding capability in all scales, i.e. if an edge with a small contrast is to be eliminated, then it should be eliminated in any scale by the corresponding threshold. This is the concept of scale-threshold consistency, which determines the relationship between a scale and a threshold. Hence, when the threshold for the slope in the coarsest scale is given, the corresponding thresholds in other scales with the same thresholding capability are also determined according to the scale-threshold relationship.

A series of measurements is conducted on an ideal step edge with a contrast of 100 so as to determine the relationship between the scale and the threshold. The RCBS fitting is applied with α ranging from 0.5 to 4.5. For each scale, the first order derivative (i.e. the slope) of the fitted curve at the position of the edge is measured. Also, the ratios of

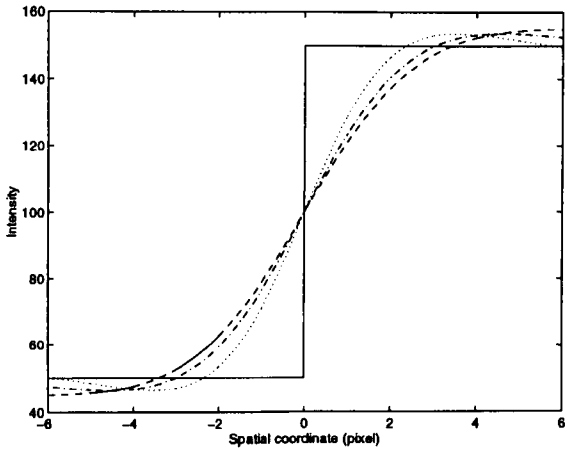


Figure 4.9: The ideal step edge and its fitted curves using the RCBS fitting with various scales. Solid line: a step edge; dotted line: $\alpha = 1$; dashdot line: $\alpha = 3$; dashed line: $\alpha = 5$.

the slopes at various scales to the slope in the coarsest scale is calculated and shown in Table 4.1. Although these ratios are measured on an edge with a contrast of 100, they can be used for edges of all contrasts. This is because a regularised fitting is a linear process (see Section 3.3.3) which results in the ratios to be independent of the contrast. In this way, the threshold with the same thresholding capability at a given scale is determined by the product of the ratio associated with that scale and the threshold used in the coarsest scale.

4.3.3 Anisotropic Diffusion

Whitaker [102, 103], and Perona and Malik [79] proposed the anisotropic diffusion. Its basic concept is that the degree of smoothing along the orientation of the edge and across the edge should be different, with a strong smoothing along the orientation of the edge to suppress noise, and a weak smoothing across the edge so as to minimise blurring of the edge.

This concept is employed in the design of MRCBS so as to locate the edge as accurately

Table 4.1: Slopes of the fitting at various scales, and the ratios of these slopes with the slope at the coarsest scale.

Scale (α)	5.0	4.5	4.0	3.5	3.0
Slope	21.991	22.545	23.172	23.892	24.734
Ratio	1.0000	1.0252	1.0537	1.0864	1.1247
Scale (α)	2.5	2.0	1.5	1.0	0.5
Slope	25.744	27.001	28.660	31.088	35.492
Ratio	1.1706	1.2278	1.3033	1.4136	1.6139

as possible. In the finest scale, before the RCBS fitting is applied along the orientation of the gradient (i.e. across the edge), an equal-weighted averaging (i.e. the Gaussian smoothing with $\sigma \rightarrow \infty$) is applied perpendicular to the orientation of the fitting. Figure 4.4(b) shows the operator kernel in the finest scale, which is also used in [17]. Since a strong smoothing is imposed along the orientation of the edge, it eliminates noise without blurring the edge.

4.3.4 The Algorithm

The pseudo codes of MRCBS are:

begin

Input (Image(x, y), threshold(coarsest scale));

Determine threshold(α)=threshold(coarsest scale) $\times$ ratio_of_slope(α);

Assign n = the size (in pixels) of the image;

For ($x = 1, 2, ...n$)($y = 1, 2, ...n$) **do**

begin

Assign α = the coarsest scale;

Do

Apply the RCBS fitting along the horizontal orientation;

Determine $f'_{horizontal}(x, y)$;

Apply the RCBS fitting along the vertical orientation;

Determine $f'_{vertical}(x, y)$;

Determine Gradient $\equiv \sqrt{(f'_{horizontal}(x, y))^2 + (f'_{vertical}(x, y))^2}$;

If Gradient < threshold(α) **then** *STOP*;

Determine EHF(2-D) $\equiv \| f''_{horizontal}(x, y) \|^2 + \| f''_{vertical}(x, y) \|^2$;

Assign α to a finer scale;

While (*NONSTOP* **and** EHF(2-D)<TEHF(2-D) **and** $\alpha > 0$)

Apply the RCBS fitting along the orientation of the Gradient;

Determine $\frac{\partial^2 f}{\partial n^2}$;

If (*NONSTOP* **and** $\frac{\partial^2 f}{\partial n^2} = 0$)

then Set EdgeMap(x, y)=*EDGE*;

else Set EdgeMap(x, y)=*NONEDGE*;

end

Output (EdgeMap(x, y));

end

4.3.5 MRCBS and Edge Focusing

In this section, the underlying principles of two multiscale schemes, MRCBS and the Edge Focusing scheme, are compared. MRCBS uses the regularisation factor α of the RCBS fitting as the scale, while the Edge Focusing scheme uses the standard deviation σ of the Gaussian pre-filter. The Gaussian pre-filter is commonly used in image processing and

computer vision because psychophysical evidences suggest that the Gaussian pre-filtering simulates the visual process of mammals (e.g. [66]). In addition, the Gaussian kernel minimises the uncertainty of the space-frequency product $\triangle x \triangle w$ (see Section 1.4) of a signal [66, 94]. Using the method of computational optimisation, Canny showed that the Gaussian approximates the pre-filter of the optimum 1-D step edge detector, where the optimisation is achieved by the suppression of noise and the localisation of edges (see Section 2.2.1). Torre and Poggio showed that the regularisation can be achieved by convolving the data with a cubic-spline filter, which is similar in shape to a Gaussian [94]. Linderberg [59] and Babaud et al. [3] have proved that the Gaussian kernel is the only kernel which does not introduce ripples in the smoothed images or cause spurious edge responses. Note that the RCBS fitting introduces ripples, which is shown as the spurious responses in the α scale space (see Section 4.2). However, as the scale is increased, the spurious responses either drift away or merge with each other. No new responses are introduced. This shows that the α scale space has the "well-behavedness" as in the σ scale space.

In the Edge Focusing scheme, a coarse scale is initially used to extract significant features. Insignificant edges and noise are either smoothed out or eliminated by a threshold. The σ scale space is used to determine the amount of deviation of an edge in a scale, and hence a series of scale (i.e. σ = 4.2, 3.85, 3.5, 3.2, 2.8, 2.5, 2.1, 1.75, 1.4, 1.0, 0.7) is heuristically chosen to ensure that the edge deviation between two successive scales are less than a pixel. By a consecutive decrement of the scale, true edge positions can be gradually traced. In the coarsest scale, a threshold is used to classify pixels with large gradients as edges. In the successive steps (at finer scales), no threshold is used, and the zero-crossings of $\frac{\partial^2 f}{\partial n^2}$ are classified as edges [5].

MRCBS is more efficient and effective than the Edge Focusing scheme for the following

reasons:

1. A fixed-size operator kernel is used to reduce the edge deviation in large scales, while retaining the ability for strong smoothing;

2. The lower bound of the scale is determined by the local EHF of the image, hence the requirement of variable blurring [5] is achieved;

3. A series of thresholds with the same thresholding capability is used in every scale to prevent noise clusters, which exist in the edge maps produced by the Edge Focusing scheme;

4. The zero-crossings of $\frac{\partial^2 f}{\partial n^2}$ are only examined in the finest scale;

5. Anisotropic diffusion is employed to impose different degrees of smoothing along the orientation of the edge and across the edge.

The differences between the Edge Focusing scheme and MRCBS are summarised in Table 4.2.

4.4 Performance Evaluations

The objective of the performance evaluations is to compare the noise-immunity and the edge-localisation of four edge detection schemes: MRCBS, the Haralick scheme [39], the Chen/Yang edge detector [17], and the Edge Focusing scheme [5]. All of these schemes adopt the natural definition of an edge [94]. The Haralick scheme, the Edge Focusing scheme and MRCBS has a variable parameter (threshold), which reflects the amount of edge details to be preserved. The Chen/Yang edge detector has two variable parameters: the scale and the threshold.

Table 4.2: Edge Focusing vs. MRCBS

Edge Focusing	MRCBS
σ Scale Space	α Scale Space
Varied Kernel Size	Fixed Kernel Size
One Threshold	Consistent Thresholds
Lower Bound of Scale = 0.7	Lower Bound Determined by EHF
Non-Variable Blurring	Variable Blurring
$\frac{\partial^2 f}{\partial n^2}$ is examined in all scales	$\frac{\partial^2 f}{\partial n^2}$ is examined in the finest scale
Isotropic Diffusion	Anisotropic Diffusion

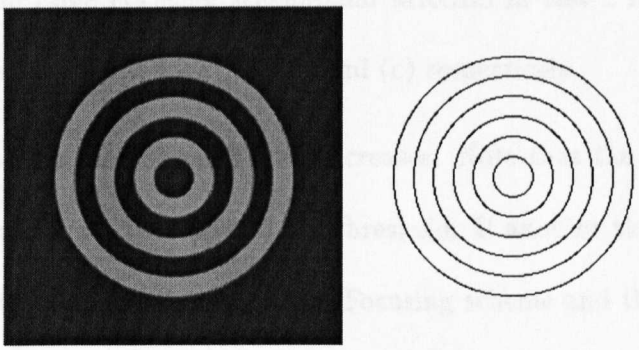
Several synthetic images, which contain edges of all orientations, are used as test images. One of the measurements of performance is the False-Correct Ratio (FCR) as in Section 3.5. However, FCR alone is insufficient to indicate the accuracy on edge localisation for a detailed comparison between edge detectors. This is because the neutral-extra and true-approximate pixels, which allow errors of one-pixel magnitude, are also considered as correctly-detected pixels. To reveal the accuracy of edge localisation, the Approximate-True Ratio (ATR) is proposed:

$$ATR = \frac{\text{number of neutral-extra pixels} + \text{number of true-approximate pixels}}{\text{number of true-positive pixels}}$$

ATR is used when the FCR's of two edge maps are similar. The lower the value of ATR, the more accurate is the edge map. If the value of FCR exceeds 1, then ATR is meaningless because the precision of the edge map should only be considered when most of the edges are correctly detected.

In the first test, a synthetic image which contains edges of all orientations and with a contrast of 100 (Figure 4.10(a)) is used. The Gaussian noise with various NSD's are

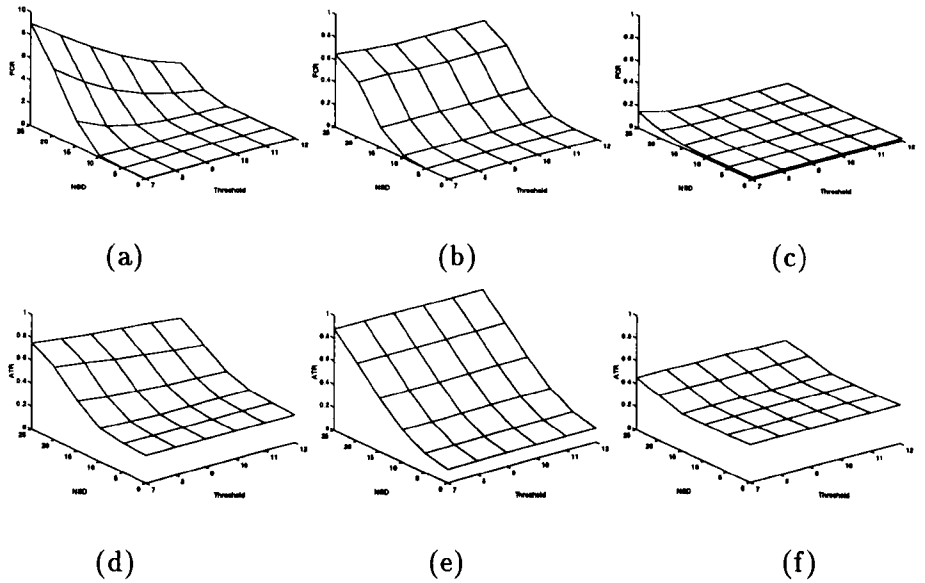
added to this image, and the FCR's and the ATR's of the edge maps produced by the four schemes are computed. Figure 4.11 shows the FCR's and the ATR's of the Haralick scheme, the Edge Focusing scheme and MRCBS using various thresholds. It shows that the performance of the Haralick scheme depends heavily on the value of the threshold when the image is noisy, and it fails in some cases when the FCR exceeds 1 (Figure 4.11(a)). The performance of the Edge Focusing scheme and MRCBS are less dependent on the threshold (Figure 4.11(b) and (c)). Since the Chen/Yang edge detector has two variable parameters, its performance has to be measured using various scales and thresholds. The results (Figure 4.12) show that the Chen/Yang edge detector has the optimum performance when the scale is 1. However, the performance in this scale depends on the value of the threshold when the image is noisy.



(a) (b)

Figure 4.10: (a) The synthetic image used in the first test; (b) The ideal edge map.

The optimum FCR's and ATR's of the four schemes in Figure 4.11 and 4.12 are shown together in Figure 4.13, to facilitate the comparison of their performances under various NSD's. MRCBS has the best performance because it has the lowest FCR's at all NSD's. The Edge Focusing scheme is less capable in dealing with noise than the Chen/Yang edge detector and MRCBS when the NSD is greater than 15. The optimum performance of the Haralick scheme is similar to MRCBS and the Edge Focusing scheme when the NSD



(a) (b) (c)

(d) (e) (f)

Figure 4.11: (a) (b) and (c): The FCR's of the edge maps which are produced by the Haralick scheme, the Edge Focusing scheme and MRCBS in test 1 respectively. (d) (e) and (f): The ATR's associated with (a), (b) and (c) respectively.

is low, but it degrades when the noise level increases. Note that the performance of the Haralick scheme depends on the value of the threshold. If another threshold is used, the performance is worse. Compared to the Edge Focusing scheme and the Haralick scheme, the Chen/Yang edge detector is less sensitive to noise. To visualise the performances of the four schemes, the optimum edge maps produced by the Haralick scheme, the Chen/Yang edge detector, the Edge Focusing scheme and MRCBS when NSD=25 are shown in Figure 4.14(a)-(d) respectively. Note that the Edge Focusing scheme produces noise clusters and zig-zag edge contours (Figure 4.14(c)) which degrade its performance.

An image which is contaminated by the Gaussian noise with NSD ranging from 0 (on the left) to 25 (on the right) is used in the second test. This image (Figure 4.15(a)) is used to examine the capability of variable blurring [5] of the four schemes. The performances of the four schemes using various thresholds are shown in Figure 4.16. Here the scale

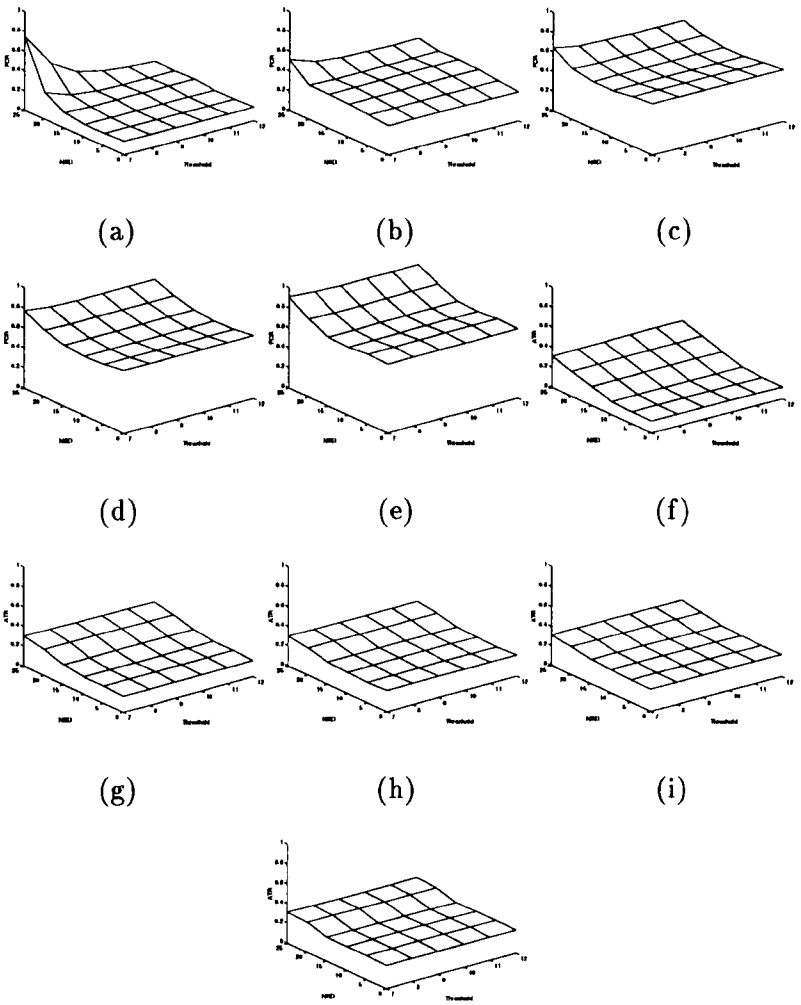
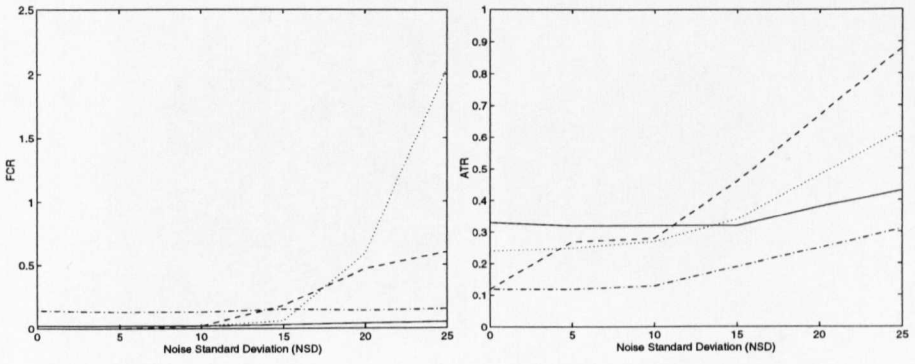


Figure 4.12: The FCR's and ATR's of the edge maps which are produced by the Chen/Yang edge detector in test 1 using various scales and thresholds. (a) $\alpha = 1$; (b) $\alpha = 2$; (c) $\alpha = 3$; (d) $\alpha = 4$; (e) $\alpha = 5$; (f)-(j) are the corresponding ATR's of (a)-(e) respectively.

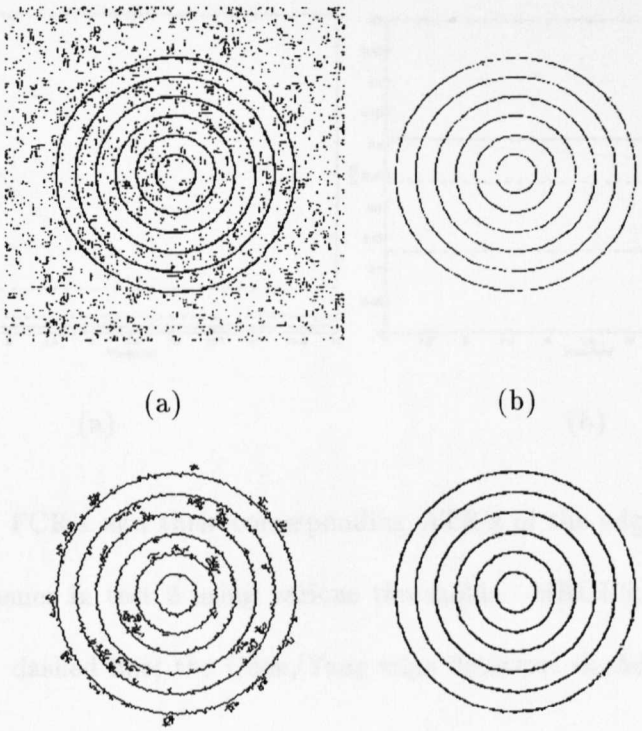


(a) (b)

Figure 4.13: The optimum FCR's and the corresponding ATR's of the edge maps produced by four different schemes in test 1 under various NSD's. MRCBS: solid line; the Edge Focusing scheme: dashed line; the Chen/Yang edge detector: dashdot line; the Haralick scheme: dotted line.

of the Chen/Yang edge detector is chosen to be 1, because this scale leads to its best performance in the first test. Figure 4.16 shows that MRCBS performs better than all the other three schemes. The performance of the Haralick scheme depends on the value of the threshold. The edge maps produced by the Haralick scheme, the Chen/Yang edge detector, the Edge Focusing scheme and MRCBS using the threshold of 7 are shown in Figure 4.17(a)-(d) respectively. (a) shows the inability of the Haralick scheme in dealing with noise. On the right half of (c), the noise clusters and the zig-zag edge contours show the "over-focused" phenomenon, which can be alleviated by variable blurring. The noise clusters can also be prevented if the thresholds with the same thresholding capability are applied in every scale. This is why MRCBS produces a better edge map (d) than the Edge Focusing scheme.

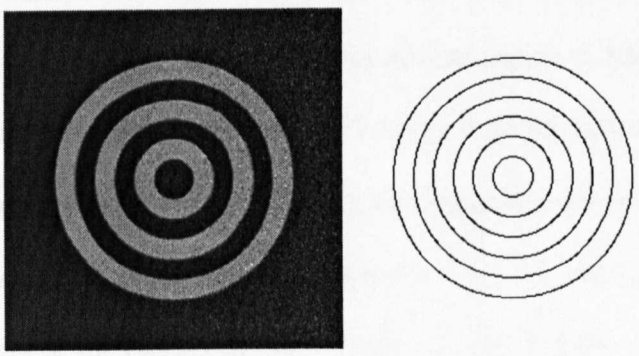
In the third test, an image of "Trevor" (Figure 4.18(a)) is used to visualise the performances of the four schemes on a real image. The main features to be observed are



(a) (b)

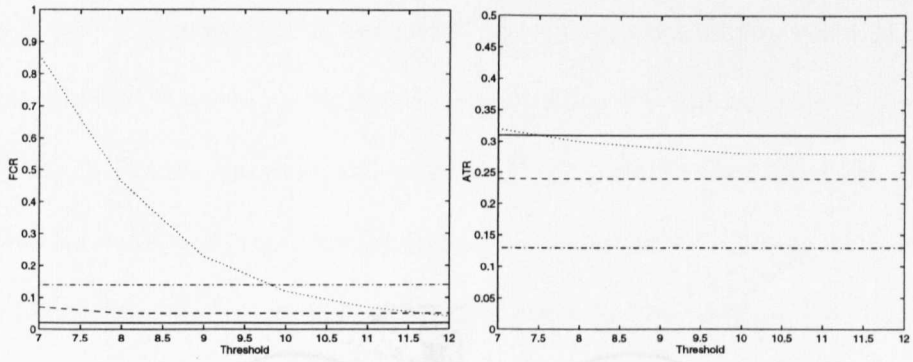
(c) (d)

Figure 4.14: (a)-(d) The optimum edge maps of the first test produced by the Haralick scheme (FCR=2.05, ATR=0.62), the Chen/Yang edge detector (FCR=0.15, ATR=0.31), the Edge Focusing scheme (FCR=0.60, ATR=0.88) and MRCBS (FCR=0.05, ATR=0.43) respectively. These edge maps are obtained when the NSD of the test image is 25.



(a) (b)

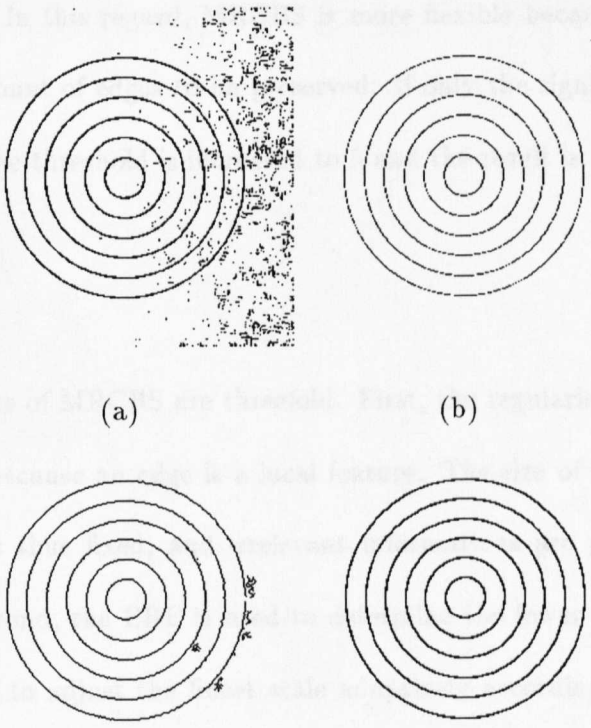
Figure 4.15: (a) The test image 2 contaminated by Gaussian noise with NSD ranging from 0 (on the left) to 25 (on the right); (b) The ideal edge map.



(a) (b)

Figure 4.16: The FCR's and their corresponding ATR's of the edge maps produced by four different schemes in test 2 using various thresholds. MRCBS: solid line; the Edge Focusing scheme: dashed line; the Chen/Yang edge detector: dashdot line; the Haralick scheme: dotted line.

the face, the tie, the wrinkles on the shirt, and the contours of the clothes. A series of thresholds is used, and the best edge maps of the Haralick scheme, the Chen/Yang edge detector, and MRCBS are shown in Figure 4.18(b), (c) and (d) respectively. All of these schemes perform satisfactory on the face and in preserving the wrinkles on the clothes. MRCBS and the Chen/Yang edge detector perform better than the Haralick scheme on the pattern of the tie. However, a large amount of trial-and-error has to be performed for the Chen/Yang edge detector before the right scale and threshold are determined. Figure 4.18(e), (f), and (g) are three of the images in the focusing process of the Edge Focusing scheme. The threshold is 2 and the scales are $\sigma = 4.2,\ 1.4$ and 0.7 respectively. Even though the contours of the clothes are well preserved, the wrinkles are eliminated due to the use of a large kernel with a strong degree of smoothing in the coarsest scale. In the final edge map (g) the "over-focused" edges are zig-zag, which distorts the facial expression. In comparison, (f) has a better facial expression than (g). When the threshold is further



(a) (b)

(c) (d)

Figure 4.17: (a)-(d) The edge maps of the second test produced by the Haralick scheme (FCR=0.86, ATR=0.32), the Chen/Yang edge detector (FCR=0.14, ATR=0.13), the Edge Focusing scheme (FCR=0.07, ATR=0.24) and MRCBS (FCR=0.02, ATR=0.31) respectively. The threshold is 7.

reduced to 1, which is a very small threshold, the details such as the wrinkles are still not detected. Instead, some other clusters of edges adjacent to the contour of the clothes appear (h), which further degrades the edge map. This shows that the Edge Focusing scheme preserves significant contours [5] such as the boundary of Trevor. All the other details are eliminated. In this regard, MRCBS is more flexible because the threshold is used to control the amount of edges to be preserved. If only the significant contours are to be preserved, then the threshold is increased to 5 and the result is shown in (i).

4.5 Summary

The major achievements of MRCBS are threefold. First, the regularisation factor of α is interpreted as a scale because an edge is a local feature. The size of the operator kernel in the α scale space is thus fixed, and irrelevant informations are precluded from the smoothing process. Second, the EHF is used to determine the lower bound of the scale, which enables MRCBS to adjust the finest scale adaptively according to the local noise level. Third, a series of thresholds with the same thresholding capability are used in their corresponding scales, which prevents noise clusters. In addition, the design of MRCBS employs the concept of anisotropic diffusion, which introduces strong smoothing along the orientation of the edge to eliminate noise. Less smoothing is imposed across the edge to minimise blurring of the edge.

Due to the above three reasons, MRCBS performs better than the Edge Focusing scheme, the Chen/Yang edge detector and the Haralick scheme. The Chen/Yang edge detector has two variable parameters (i.e. the scale and the threshold) to be determined, and MRCBS provides a means to determine the scale adaptively according to the local noise level, while the variable parameter of the threshold is used for the adjustment to

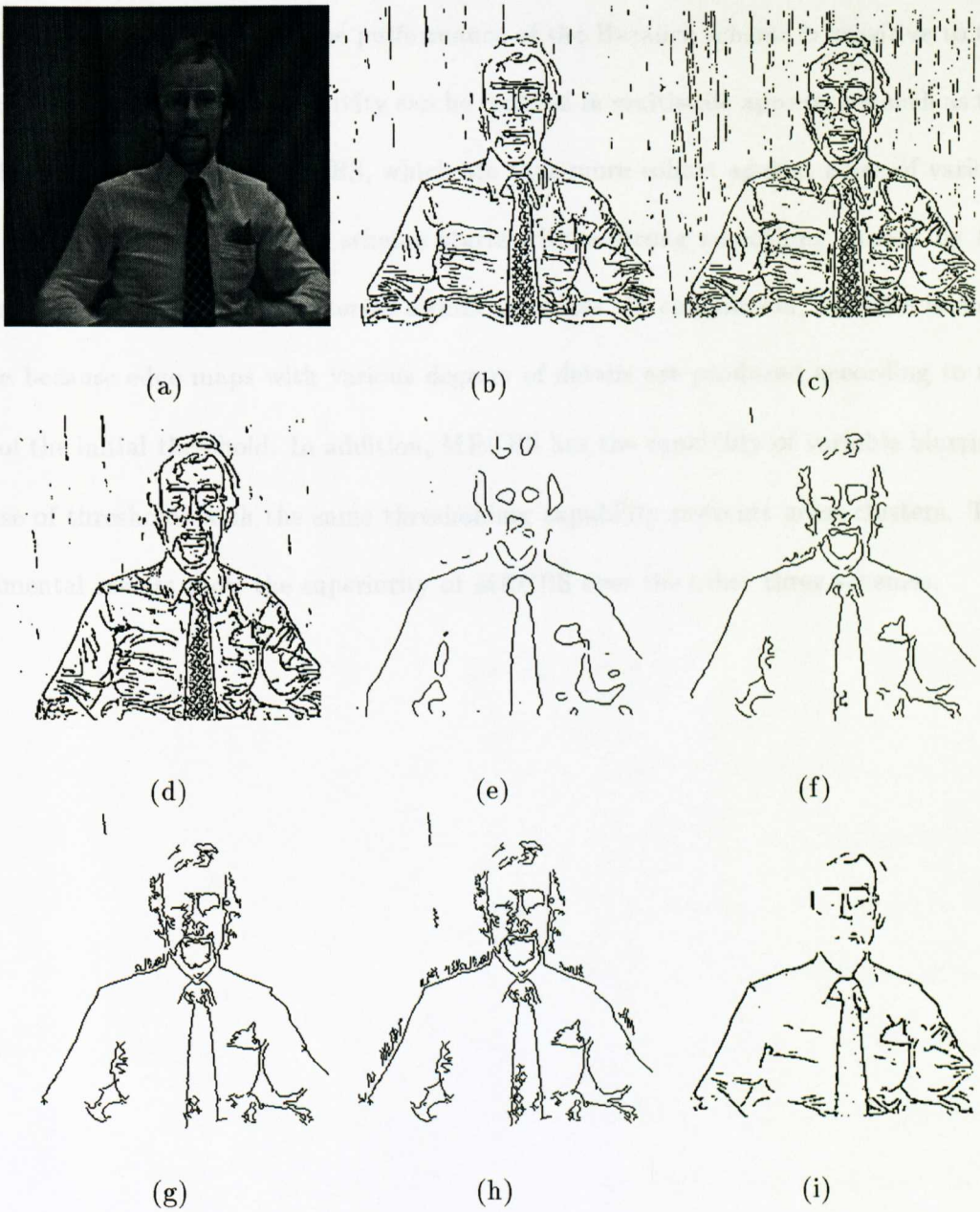


Figure 4.18: (a) The real image of "Trevor". (b), (c) and (d): The optimum edge maps produced by the Haralick scheme (threshold=5), the Chen/Yang edge detector (α=0.1, threshold=4) and MRCBS (threshold=2) respectively. (e), (f) and (g) Three of the edge maps in the focusing process of the Edge Focusing scheme, where (G) is the final edge map. The threshold is 2 and the scales are σ = 4.2, 1.4 and 0.7 respectively. (h) The final edge map (σ = 0.7) produced by the Edge Focusing scheme using a threshold of 1. (i) The edge map produced by MRCBS using a threshold of 5.

satisfy different requirements. The performance of the Haralick scheme is sensitive to the value of the threshold. This sensitivity can be reduced in multiscale approaches such as the Edge Focusing scheme and MRCBS, which are thus more robust against noise of various levels. Since the Edge Focusing scheme starts with a strong smoothing caused by the large operator kernel, it only preserves significant edges. In comparison, MRCBS is more flexible because edge maps with various degrees of details are produced according to the value of the initial threshold. In addition, MRCBS has the capability of variable blurring. The use of thresholds with the same thresholding capability prevents noise clusters. The experimental results show the superiority of MRCBS over the other three schemes.

Chapter 5

Texture Focusing

5.1 Statistical Texture Segmentation

Texture segmentation is a basic aptitude of human vision in distinguishing an object from the background using the surface texture of the object. It is thus an important issue in computer vision. A natural texture is tessellated by primitives known as texels [47] or textons [48] in a certain structure. This has resulted in several structural-based texture analysers where symbolic descriptions are used to represent the pattern of the tessellation of these primitives [99]. In addition to these structural-based methods, schemes such as the co-occurrence matrices [37] which are based on statistical decision rules have also been investigated.

Bouman *et al.* define texture segmentation as the process which divides an image into regions with distinct statistical distributions of the image grey levels [11, 12]. This definition is adopted in this thesis, where a textural feature is defined as the set of statistics which represents the textural contents of a block of image pixels. The textural feature space, a Cartesian space where each coordinate represents a statistic of the textural content, is the basis of the segmentation.

5.2 Uncertainty, Multiresolution and Adaptivity

An adequate texture segmentation scheme needs to employ all the local, global and contextual information of the image. The reasons being: (1) the boundary between different textures has to be located accurately; (2) the textural content is estimated within a (global) spatial extent which corresponds to a block of pixels; and (3) neighbouring blocks are likely to contain the same texture [95]. According to the uncertainty principle of the joint space-frequency analysis (see Section 1.4), high resolutions in both the spatial domain (correspond to a small window) and the feature space (correspond to a narrow frequency bandwidth) cannot be achieved in a single process. A large spatial extent improves the estimation of the textural content at the expense of the spatial resolution of the segmentation map. For example, Figure 5.1(a) contains two textures, and the boundary between the two textures are shown in Figure 5.1(b). The variances of Figure 5.1(a) are measured using block-shaped windows of different sizes. The values of the variances are shown as the grey-level in Figure 5.1(c) with size = 7×7 and Figure 5.1(d) with size = 25×25. In Figure 5.1(c), edges which depict the details of texels are enhanced. This is similar to the edge map in Figure 5.1(e) produced by the Laplacian of Gaussian (LoG) edge detector (see Section 2.2). Both Figure 5.1(c) and Figure 5.1(e) fail to locate the boundary between the two textures. Conversely, two textures are roughly separated in Figure 5.1(d) at the expense of the spatial resolution. This test illustrates two points: first, the spatial resolution, which corresponds to the unit block size, influences the result of the segmentation significantly; and second, a single-scale local operator such as LoG is inadequate for texture segmentation. This is because no significant local features exist at the boundary of the texture (see Figure 5.1(f)). The above observations justify that texture segmentation is a multiresolution task where both global and local information

are important. Note that if a statistic is computed using a window which is smaller than a texel, it will only represent a portion of this texel and give a misleading estimation of the texture. More precisely, the resolution of the segmentation map are upper-limited by the size of the largest texel.

A few multiresolution schemes have been introduced in recent years (e.g. [15, 97]). Chang and Kuo use the tree-structured wavelet transform to analyse images with various textures in various channels [15]. These channels are adaptively chosen according to the energy distribution of the texture in the scalogram of the image. Wilson *et al.* proposed the Multiresolution Fourier Transform (MFT) which uses different resolutions to represent the different portions of an image. Each block in this representation satisfies a pre-defined hypothesis (e.g. the existence of a linear feature) [106]. The tree-structured wavelet transform and MFT are two examples of adaptive image processing, where the adaptivity of the tree-structured wavelet transform lies in the feature domain, and that of MFT lies in the spatial domain. MFT has been implemented for edge detection [106] and curve extraction [14]. However, the implementation of MFT for texture segmentation is still in progress [55].

A few attempts have also been made on the application of the Markov Random Fields (MRF) for texture segmentation (e.g. [2]). Geman and Geman [33] demonstrate the relationship between the Markov random field and the Gibb's distribution, and employ a stochastic relaxation method of simulated annealing [52] to optimise the posterior distribution for restoring images. Following the above approach, Muzzolini *et al.* employ the simulated annealing in a multiresolution framework for texture segmentation [71, 74]. A quad-tree structure is used to represent an image, where the level of the quad-tree corresponds to the image resolution. The simulated annealing is used to determine the

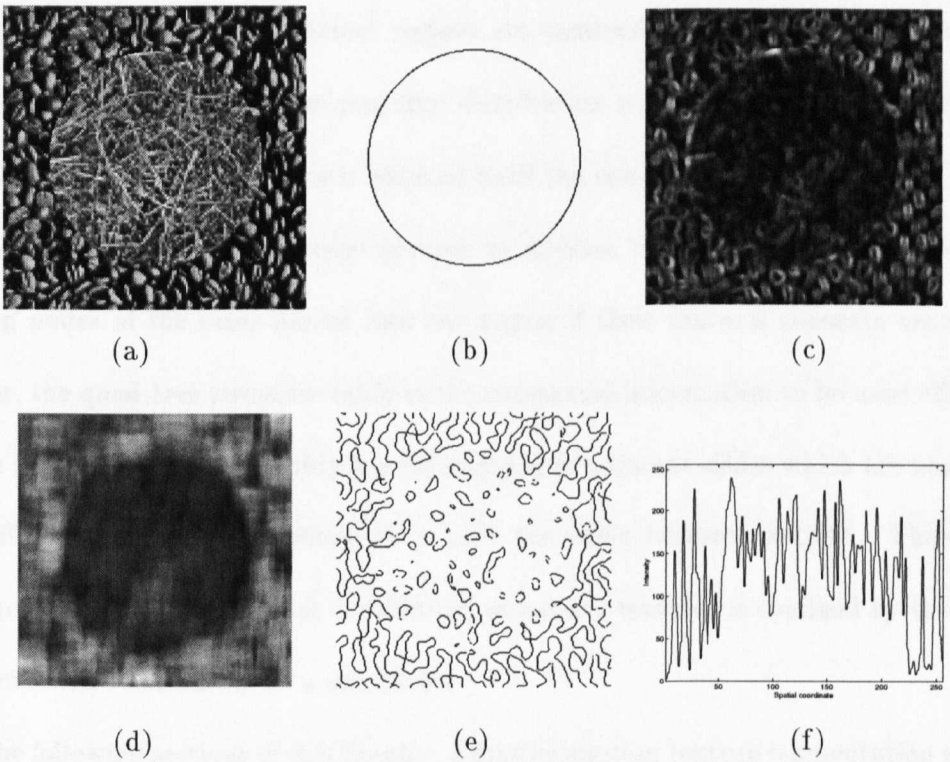


Figure 5.1: (a) An image with two textures- nuts and straw. (b) The boundary between the two textures. The variances of (a) using windows: (c) of size = 7 (pixels) and (d) size = 25 (pixels). (e) The edge map of (a) produced by the Laplacian of Gaussian scheme. (f) The luminance profile along the central line of (a).

increment or decrement of the resolution of a randomly chosen leaf node in the quad-tree (i.e. the split and merge process [71]). If a leaf node contains several features, then this node is likely to be split; if the four leaf nodes associated with the same parent node have the same feature, then they are likely to be merged as one node. The likelihood of split or merge is determined by the Metropolis probability (see Section 5.3.4). At the start of the segmentation, classes of different regions are randomly altered and the resolution is continuously adjusted. When the posterior distribution is gradually increased through an iterative process, the randomness is reduced until the computation is terminated.

Muzzolini *et al.* use the merge process to address the contextual information, i.e. grouping nodes of the same parent into one region if their textural contents are similar. However, the quad-tree structure inhibits the contextual information to be used efficiently because it precludes the possibility for the merging of adjacent nodes which are associated with different parents even though they have the same textural content. There is no reason to believe that the spatial distribution of natural textures is confined by the logical parent-children relationship of a quad-tree.

In the following sections of this chapter, a multiresolution texture segmentation scheme, referred to as Texture Focusing, is proposed. This scheme is computationally simpler than the various MRF-related methods, and achieves high resolutions in both the spatial and featural domains using a multiresolution process. The quad-tree is used without the merge process so as to avoid the rigorous constraint between nodes of the quad-tree. A split-and-fix process is also introduced, which employs the contextual information as an indication for the adequate resolution in each region of the segmentation map. In addition, texture focusing possesses both the merits of adaptivity of MFT and tree structured wavelet transform in the spatial and featural domains respectively.

5.3 The Design of Texture Focusing

5.3.1 Texture Characterisation

When the image resolution is fixed, the task of texture segmentation is a labelling task because each block of pixels is labelled according to its textural content. A labelling task comprises two stages: to determine a set of characteristic measures as a feature, and to partition the feature space [31]. Due to the diversity of natural textures, the selection of a feature for texture segmentation is a difficult task, and there seems to be no textural features which can characterise all known textures. Tamura *et al.* conducted a series of psychophysical tests on the human vision to determine an adequate textural feature [91]. They propose six basic elements of textures including coarseness, contrast, directionality, line-likeness, regularity and roughness [91]. Francos *et al.* propose a texture model which comprises three components: the purely-indeterministic field, the harmonic field and the generalised evanescent field [29]. In the segmentation scheme proposed by Muzzolini *et al.* [74], samples of the image are used to characterise the texture. A textural feature is thus determined and used for the segmentation. Similarly, the unsupervised texture segmentation scheme proposed by Hofmann *et al.* comprises the modelling and the optimisation stages [42], where the modelling stage corresponds to the process of texture characterisation.

The number of dimensions of a textural feature space is a trade-off between the ability to discriminate similar textures and the computation cost [72]. The more textures to be distinguished, the more dimensions are required. For the sake of simplicity, each of the statistics of mean, standard deviation and quasi skewness of a block of image pixels serves as a dimension of the Cartesian textural feature space. These statistics are defined as follows:

$$mean = \frac{\sum grey\ level}{n},$$

$$standard\ deviation = \sqrt{\frac{\sum(grey\ level - mean)^2}{n}},$$

$$quasi\ skewness = \left[\frac{\sum(grey\ level - mean)^3}{n}\right]^{\frac{1}{3}}.$$

Thus, a textural feature $\equiv$ {mean, standard deviation, quasi skewness }, which is a vector in the 3-D textural feature space.

5.3.2 Spatio-Featural Mutual Focusing

In addition to the complexity in characterising a natural texture, Andrey and Tarroux point out a dilemma which exists in the second stage of texture segmentation, that the value of the textural feature, which is required to partition the feature space, cannot be accurately determined until the image has been adequately segmented [2]. If the size of the texel is known and used as the window size for the analysis, then various clustering methods can be used to group blocks with the same texture into one region, and hence solve the above dilemma. However, the size of the texel is very difficult to estimate. Furthermore, these clustering methods involve either an iterative process or a parameter to be given heuristically. For example, the starting point in the cluster-seeking algorithm, the maximin distance ratio in the maximin-distance algorithm, and the number of clusters (i.e. the value of K) in the K-means algorithm have to be given heuristically [95]. To cope with the situations when the size of the texel is unknown, a quad-tree image structure is employed in the multiresolution clustering process of Texture Focusing. In addition, the root of the quad-tree, which corresponds to the entire image, is a natural starting point for texture segmentation.

The coarse-to-fine process is commonly adopted by various multiresolution schemes. For example, the edge focusing algorithm traces the locations of salient edges from a coarse spatial resolution to its finest resolution (see Chapter 4 and [5]). In texture segmentation, the locations of boundaries between textures also need to be focused, which justifies the use of the coarse-to-fine process for spatial focusing. At the same time, the textural content also needs to be accurately determined for the segmentation. However, the resolution of the feature space is low in the fine spatial resolution as indicated by the uncertainty principle. The estimated feature is thus unreliable. This is why the feature focusing is introduced in conjunction with the spatial focusing to construct the spatio-featural mutual focusing in the coarse-to-fine multiresolution clustering scheme. The basic idea is that a good spatial estimation of the boundary results in a good estimation of the textural feature, and vice versa. The coarse-to-fine process increases the spatial estimation of the boundary as the level of the quad-tree increases. At each level, featural focusing re-estimates the texture content according to the segmented regions of the current level, which is spatially more accurate than those of the previous level.

To begin with, the textural feature (i.e. {mean, standard deviation, quasi skewness }) of the entire image (i.e. the root of the quad-tree) is computed and serves as the cluster centre for the segmentation at the next level. Also, the class of the root node is propagated temporarily to its children nodes. The textural feature of each node at the next level is then computed. If the textural feature of a node is in the vicinity of the cluster centre of its temporary class, then the temporary class is assigned as the class of the current node, thus preserving the global information. (The vicinity of a class centre is determined by the Linear Temperature-Varying Probability which will be introduced in Section 5.3.4.) Otherwise, the scheme searches for another cluster centre in the feature space which is the closest to the feature of the current node. If the feature of the current node is not

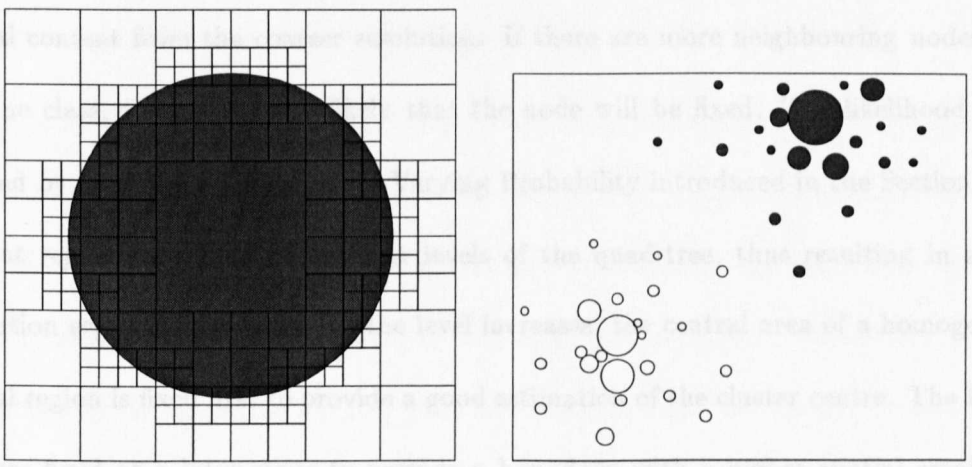
in the vicinity of any class centre, then a new cluster centre is created using the feature of the current node. Note the number of classes (textures) is flexible which reflects the number of textures detected at each level. Thus, a dynamic chain is designed to store the information of the classes (e.g. the cluster center) for the flexibility of the scheme (see Section 5.3.5).

The image is roughly segmented by the labelling of nodes into different classes (textures). The resolution of the roughly segmented image is determined by the block size of the node. All the cluster centres are re-computed according to the current segmentation map (i.e. feature focusing). Since the current segmentation map is more accurate than the map at the previous level, the cluster centres, determined on large regions, are better estimated. If all the nodes originally belonging to a class are re-assigned to other classes, then this class is eliminated from the dynamic chain. This whole process is repeated at the next levels until the upper bound of the resolution is reached. In this way, the boundary is focused to its best resolution and the textural content within this boundary becomes most accurately determined, i.e. both the global and the local information are effectively used.

5.3.3 Split and Fix

In texture focusing, the block size decreases as the level of the quad-tree increases. Although the cluster centres are determined on large regions, the estimation of the textural content of each node becomes less and less accurate as the level increases. In addition, the upper bound of the resolution, which corresponds to the size of the texel, is very difficult to determine. The split-and-fix paradigm, which results in an adaptive multiresolution representation of an image similar to MFT, is proposed for solving the above problems. The 'split' represents a coarse-to-fine process. In this multiresolution representation, the

central areas of homogeneous textures are analysed using larger blocks, while the borders of textures are analysed in the sub-texel resolutions so as to achieve better estimation of the boundary (see Figure 5.2(a)). The feature space corresponding to the multiresolution representation is shown in Figure 5.2(b), where the radius of the circle represents the block size of a region. The central area of a homogeneous texture determines an initial cluster centre in the feature space. The sub-texel segmentation is assumed to be credible if the cluster centre has already been accurately determined using the central area of a homogeneous texture.



(a) (b)

Figure 5.2: (a) A multiresolution representation of an image. The two textures are depicted as grey and white. The central regions of homogeneous textures are represented using large windows, whereas the border regions of textures are represented using small windows. (b) The feature space of (a), where large circle represent the texture feature of a large region in (a).

The contextual information is used in the 'fix' process to determine the adequate resolutions for different regions in an image. In the image context, the features of neighbouring blocks are similar to that of the central block [95]. To use the contextual information, the

adjacent 8, 5 and 3 blocks are defined respectively as the neighbouring blocks of a block in the central, the border and the corner area of an image (Figure 5.3). At each level, every node of the quad-tree and its neighbours are classified using cluster centres in the textural feature space. If the class of a node is identical to some of the neighbouring nodes, then this node is assumed to be situated within a homogeneous texture region, and the class of this node is likely to be fixed, i.e. the class is likely to be assigned to all its children nodes without an estimation on the textural feature in the higher resolutions. The above procedure, referred to as the 'fix' process, is used to preserve a credible estimation of the textural content from the coarser resolution. If there are more neighbouring nodes with the same class, then it is more likely that the node will be fixed. The likelihood is determined by the Linear Temperature-Varying Probability introduced in the Section 5.3.4. Different regions are fixed at different levels of the quad-tree, thus resulting in a multiresolution segmentation map. As the level increases, the central area of a homogeneous textural region is fixed first to provide a good estimation of the cluster centre. The border areas are fixed at a later stage to provide a boundary with a higher spatial resolution. The split-and-fix process is repeated until the pixel level is reached, thus resulting in a sub-texel segmentation.

5.3.4 Linear Temperature-Varying Probability

The simulated annealing is commonly used to achieve the global optimisation of a complex task where the state space is too large to be examined thoroughly [52]. In simulated annealing, the Metropolis probability plays an essential role in determining the probability of a change of state $P(change)$, i.e.

$$P(change) = \begin{cases} 1 & if \triangle U \leq 0; \\ e^{-\frac{\triangle U}{T}} & else, \end{cases} \tag{5.1}$$

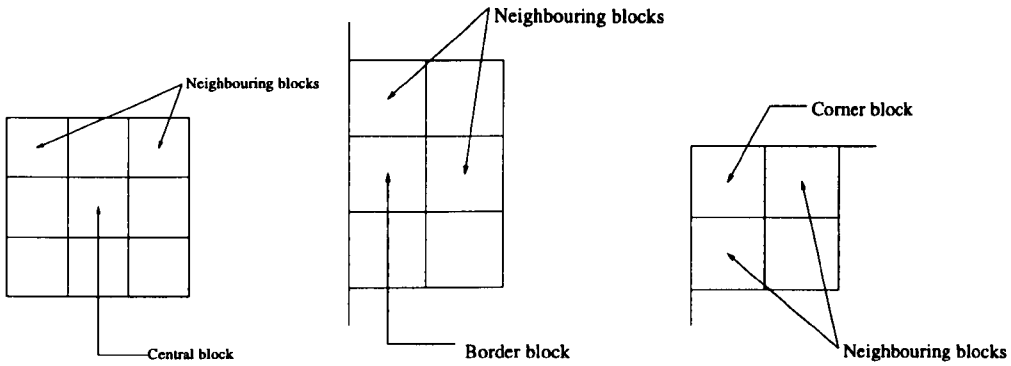


Figure 5.3: The neighbouring blocks of a block in the (a) central area, (b) border area, (c) corner area of an image.

where T is the temperature and $\triangle U$ is the increment in the optimisation function caused by a random change of state. If $\triangle U \leq 0$, $P(change) = 1$, the state is changed to reduce the optimisation function. Otherwise, $P(change)$ is an exponential function of $\triangle U$ and T. The principle behind the simulated annealing is that the global parameter T controls the probability of change so that it decreases continuously. In the early stage of the computation, the state is easily adjusted to search for the global minimum. Thereafter, the probability of change is reduced. As $T \rightarrow 0$, $e^{-\frac{\triangle U}{T}} \rightarrow 0$, and the state is assumed to have converged to the global minimum.

The concept of using a global parameter to control the probability of change is useful in providing the adaptivity in texture focusing, where the probability of a change in the state is adjusted in different stages of the process. The complex computation caused by the exponential in the Metropolis probability motivates a simpler probability function, referred to as the Linear Temperature-Varying Probability (LTVP), which uses the first

two terms of the Maclaurin series of $e^{-\frac{\Delta U}{T}}$, i.e. $1-\frac{\Delta U}{T}$. Therefore, equation (5.1) becomes:

$$LTVP(change) = \begin{cases} 1 & if \triangle U \leq 0; \\ 0 & if \triangle U \geq T; \\ 1 - \frac{\Delta U}{T} & else. \end{cases}$$

The comparison in the use of the Metropolis probability and LTVP is illustrated in Figure 5.4. Texture focusing is designed as a non-optimisation scheme (an optimisation scheme must have an explicitly defined energy function), therefore LTVP is used to determine the probability of assigning a class to a node during the 'split' process (i.e. $LTVP(assign)$) as well as the probability of fixing a central block during the 'fix' process (i.e. $LTVP(fix)$). LTVP is a linear function of ΔU which starts from the change of a state ($LTVP(change) = 1$) to the retainment of a state ($LTVP(change) = 0$) within a range of ΔU determined by T. This linear range is referred to as a *margin* and an *increment* in the implementation of $LTVP(assign)$ and $LTVP(fix)$ respectively. Both probabilities represent the retainment of a state rather than the change of a state as in the simulated annealing. Therefore, T needs to be increased to reduce the probability of change. In the quad-tree coarse-to-fine algorithm, the level of the quad-tree L is a natural choice of T, i.e. $T \equiv L$, where $L \in \mathbf{Z}$ and $L = 0$ denotes the root of the quad-tree.

In the 'split' process, the assignment of a class to a node is determined by the Euclidean distance *dist* between the class centre and the textural feature of the node in the textural feature space. If the textural feature of a node is identical to the class centre, then $LTVP(assign) = 1$. If the distance between them is greater than a margin, then $LTVP(assign) = 0$. Otherwise, $LTVP(assign)$ decreases linearly from 1 to 0 (Figure 5.5(a)). According to LTVP, a margin is proportional to the global parameter L, i.e.

$$margin = (basis\_margin) \times L.$$

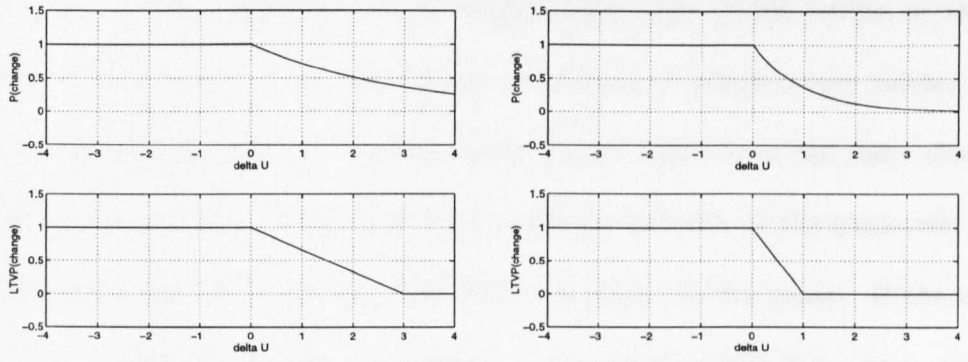
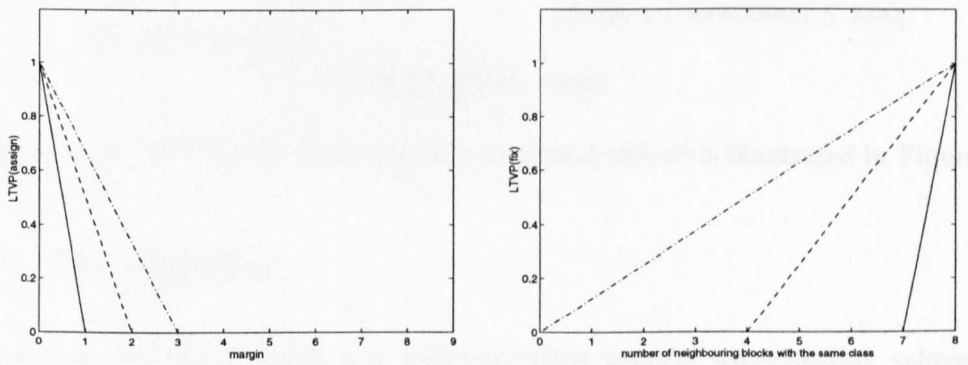


Figure 5.4: Upper Left: The $P(change)$ determined by the Metropolis probability when T=3; Lower Left: $LTVP(change)$ when T=3; Upper Right: The $P(change)$ determined by the Metropolis probability when T=1; Lower Right: $LTVP(change)$ when T=1.

The *basis_margin* is determined by the product of a parameter *margin_ratio* (which is determined in the experiment) with the length of the feature vector of the root node, i.e.

$$basis\_margin = (margin\_ratio) \times \|feature(root)\|.$$



(a) (b)

Figure 5.5: (a) $LTVP(assign)$ at various levels. L=1: solid line; L=2: dashed line; L=3: dashdot line. (b) $LTVP(fix)$ at various levels. L=3: solid line; L=6: dashed line; L=10: dashdot line.

The 'fix' process starts from level 3 of the quad-tree because there are insufficient

nodes lying in the central area of the image rather than at the border or the corner area at level 0, 1 and 2. Let *nnb* denote the number of neighbouring blocks, i.e. 8, 5 or 3; *no* denote the number of neighbouring blocks which have the same class as the central block; and *total_level* denote the number of the levels in the quad-tree, which is determined by $log_2(w)$ where w is the width (in pixels) of the image. If the classes of all of the neighbouring blocks are identical to the central block (i.e. $no = nnb$), then $LTVP(fix) = 1$. If $nnb - no$ is greater than the linear range of LTVP (referred to as an *increment*), then $LTVP(fix) = 0$. The linear range *increment* is proportional to $L - 2$, i.e.

$$increment = (L-2) \times \frac{nnb}{(total\_level) - 2},$$

$L - 2$ is chosen instead of L because the 'fix' process starts from level 3. When $L = total\_level$, $increment = nnb$. Therefore, $LTVP(fix)$ is determined by the following equation:

$$LTVP(fix) = \begin{cases} 0 & if\ (no + increment) \leq nnb; \\ \frac{no - nnb + increment}{increment} & else. \end{cases}$$

An example of $LTVP(fix)$ when $total\_level$=10 and nnb=8 is illustrated in Figure 5.5(b).

5.3.5 The Algorithm

In summary, texture focusing is a multiresolution texture segmentation scheme which alleviates the uncertainty inherent in the task by focusing both the spatial boundaries and the estimation of the textural content. The split-and-fix process, which employs the contextual information of an image, is incorporated in texture focusing to result in a multiresolution structure of the segmentation map. The $LTVP$ is used to determine the probabilities of the assignment of a class to a node ($LTVP(assign)$), and the fixing of a node ($LTVP(fix)$). A random number ($0 \leq random\_no \leq 1$) is thus compared with

$LTVP(assign)$ or $LTVP(fix)$ to decide the execution/suspension of an assignment or a fixing process. In the implementation, two quad-trees are used respectively to contain the textural feature (denoted as feature (L, node)) and the class (denoted as class (L, node)) of each node. A dynamic chain is also used to contain the class information (e.g. *cluster_centres*) of all the classes. The data structures and pseudo codes are as follows:

Definitions of data structures

quad-tree: feature (L, node)={*mean, standard deviation, quasi skewness*};

/* the textural feature of a node at level L */

quad-tree: class (L, node)={*class_pointer, fixed*};

/* the class of a node at level L */

dynamic chain: class_table (*class_pointer*)={*cluster_centre, class_number*};

/* a table of cluster centres and their corresponding number of nodes which belong to the class */

Pseudo Codes

begin

Input (Image, *margin_ratio*);

Assign w = the width (in pixels) of the image;

Assign $total\_level = log_2$ (w);

Determine feature (0, root);

Assign $basis\_margin = margin\_ratio \times ||feature\ (0, root)||$;

Assign $margin = basis\_margin$;

For L=1 to *total_level* **do** /* coarse to fine */

begin

/* split */

Propagate all the class (L-1, parent) to their children nodes' class (L, children);

For every node at level L **do**

begin

If (class (L, node) $\neq$ fixed)

begin

Determine p1 = feature (L, node), p2 = class_table (class (L, node));

Determine *dist* = distance (p1, p2);

Determine $LTVP(assign)$ using *dist* and *margin*;

If ($random\_no \geq LTVP(assign)$)

begin /* search for a new class */

For every class in the class_table **do**

Determine *dist* = distance (*cluster_centre*, feature (L, node));

Determine the closest_class such that *dist* is minimised;

Determine $LTVP(assign)$ using *dist* of the closest_class and *margin*;

If ($randon\_no \leq LTVP(assign)$)

Assign class (L,node) = closest_class;

else

Create new_class = { feature (L, node), 1 };

Append new_class to class_table;

Assign class (L, node) = new_class;

end

end

end

Assign *margin* = *margin* + *basis_margin*;

/* fix */

If (L > 2)

begin

Determine *increment*=(L-2) $\times \frac{nnb}{(total\_level)-2}$;

For every node at level L **do**

begin

/* use contextual information */

Determine *no* = *nnb* x P(class (neighbour) = class (the central node));

Determine $LTVP(fix)$ using *increment* and *no*;

If ($random\_no \leq LTVP(fix)$)

class (L,node)=fixed; /* update un-fixed pixels to fixed */

end

end

For every class in the class_table **do** /* feature focusing */

begin

If (*class_number*=0)

Eliminate the current class from the class_table;

else

Determine *cluster_centre* using { feature (L, node) | class (L, node) = current class };

end

end

Produce Segmentation Map according to class (total_level, node);

Output (Segmentation Map)

end

5.4 Performance Evaluations

Three experiments on images with two natural textures are designed to evaluate the performance of the proposed texture focusing scheme. The spatial location of the boundary between the two textures of each image is known precisely to enable a quantitative evaluation of the performance. In addition, a performance measurement, referred to as the percentage of the correct segmentation (PCS), is defined as follow:

$$PCS = \begin{cases} 0 & if\ ODC = IDC; \\ \frac{NODC+NIDC}{w^2} \times 100 & else, \end{cases}$$

where w is the width (in pixels) of the image; ODC and IDC represent the Class with Dominant (largest) number of pixels Outside and Inside the boundary, respectively. NODC is the Numbers of pixels outside the boundary which is also classified as ODC. Similarly, NIDC is the Numbers of pixels inside the boundary which is also classified as IDC. The range of PCS is from 0 to 100, the higher the PCS the better the segmentation. The PCS of a perfect segmentation map is 100.

The first experiment is designed to examine the performance of texture focusing using different values of *margin_ratio*, the only parameter in the scheme. The test image (Figure 5.6(a)) is composed of two textures of metal and straw, and the boundary between the two textures is a circle with a radius of 80 pixels (Figure 5.6(b)). The size of the image is 256x256 pixels (i.e. w=256). The PCS's of the segmentation map using various *margin_ratios* are shown in Table 5.1 and Figure 5.7. Some of the segmentation maps are shown in Figure 5.6(c)-(f), where regions with different textures are visualised using different grey levels. When the *margin_ratio* is between 0.23 to 0.40, the PCS's of the segmentation results are greater than 95, thus showing that the performance of texture focusing is insensitive to a wide range of *margin_ratios*. When the *margin_ratio*

is too small, each region is sub-divided into several regions (Figure 5.6(c)). This is because the scheme picks up the variation in the illumination from one area of the image to another. The influence of the variation in luminance will be further investigated in the third experiment. When the *margin_ratio* is too large (i.e. ≥ 0.41), the scheme is unable to distinguish the difference between the two textures, and thus results in the PCS of 0 (Figure 5.6(f)).

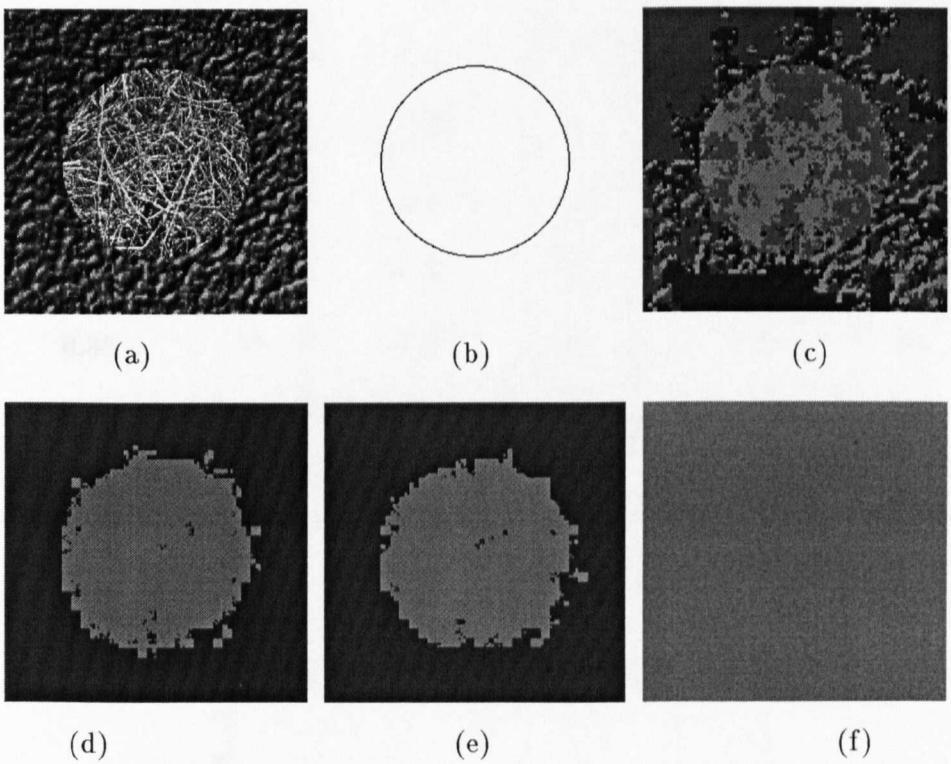


Figure 5.6: (a) An image with two textures- metal and straw. (b) The boundary between the two textures. The segmentation map of (a) using a : (c) *margin_ratio* of 0.17 (PCS=48); (d) *margin_ratio* = 0.32 (PCS=96); (e) *margin_ratio* = 0.37 (PCS=97); and (f) *margin_ratio* = 0.45 (PCS=0).

The second experiment is designed to show the intermediate states of the segmentation (Figure 5.8) and the fixing process (Figure 5.9) at each level. Figure 5.6(a) is used again as the test image, and the *margin_ratio* is 0.37 because it results in the highest PCS

Table 5.1: PCS's of segmentation maps of Figure 6(a) using various *margin_ratios*.

margin_ratios	PCS's	*margin_ratios*	PCS's	*margin_ratios*	PCS's
0.15	42	0.16	57	0.17	48
0.18	45	0.19	82	0.20	81
0.21	91	0.22	93	0.23	95
0.24	96	0.25	96	0.26	97
0.27	96	0.28	97	0.29	97
0.30	97	0.31	97	0.32	96
0.33	96	0.34	96	0.35	96
0.36	96	0.37	97	0.38	96
0.39	96	0.40	95	0.41	0
0.42	0	0.43	0	0.44	0

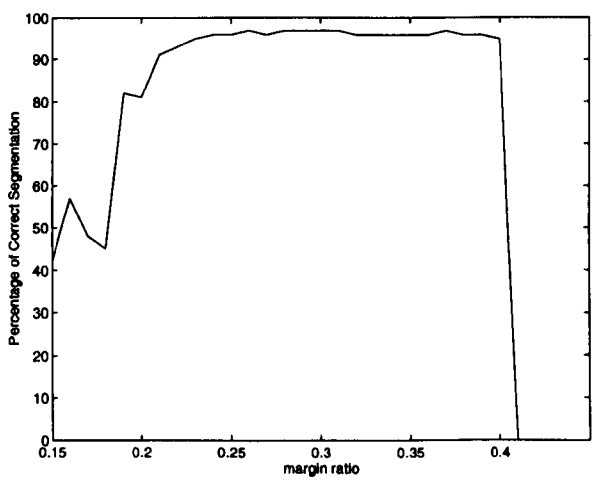


Figure 5.7: The percentages of the correct segmentation (PCS's) of the segmentation results of Figure 5.6(a) using various *margin_ratios*.

(97) in the first experiment. At level 1, the four nodes have similar textural features and are thus assigned to the same class (Figure 5.8(a)). Since the image is divided into two regions at level 2, the scheme begins to focus the estimation of the textural features using the rough segmentation. The PCS's and the percentage of the fixed area at each level are shown in Table 5.2. The PCS increases rapidly from 0 to 90 as the level increases to level 3, and then increases up to 97 at levels 6, 7 and 8. The PCS does not increase any further after level 6 because the resolutions at the higher levels are smaller than a texel. For example, each pixel is classified individually at level 8, and a pixel contains only the grey level information which is not sufficient for the estimation of texture contents. However, the finer resolutions result in smoother boundaries in the segmentation maps.

Table 5.2: PCS's and the percentage of the fixed region at various levels.

Level	PCS's	percentage of fixed region
1	0	0.0
2	81	0.0
3	90	48.4
4	93	73.8
5	95	88.4
6	97	95.2
7	97	98.3
8	97	99.4

Figure 5.9 shows the intermediate fixed regions at different levels. The fixed region is represented by grey while the un-fixed by black. The 'fix' process starts from level 3, and the fixed regions grow as the level increases. The central area of a homogeneous

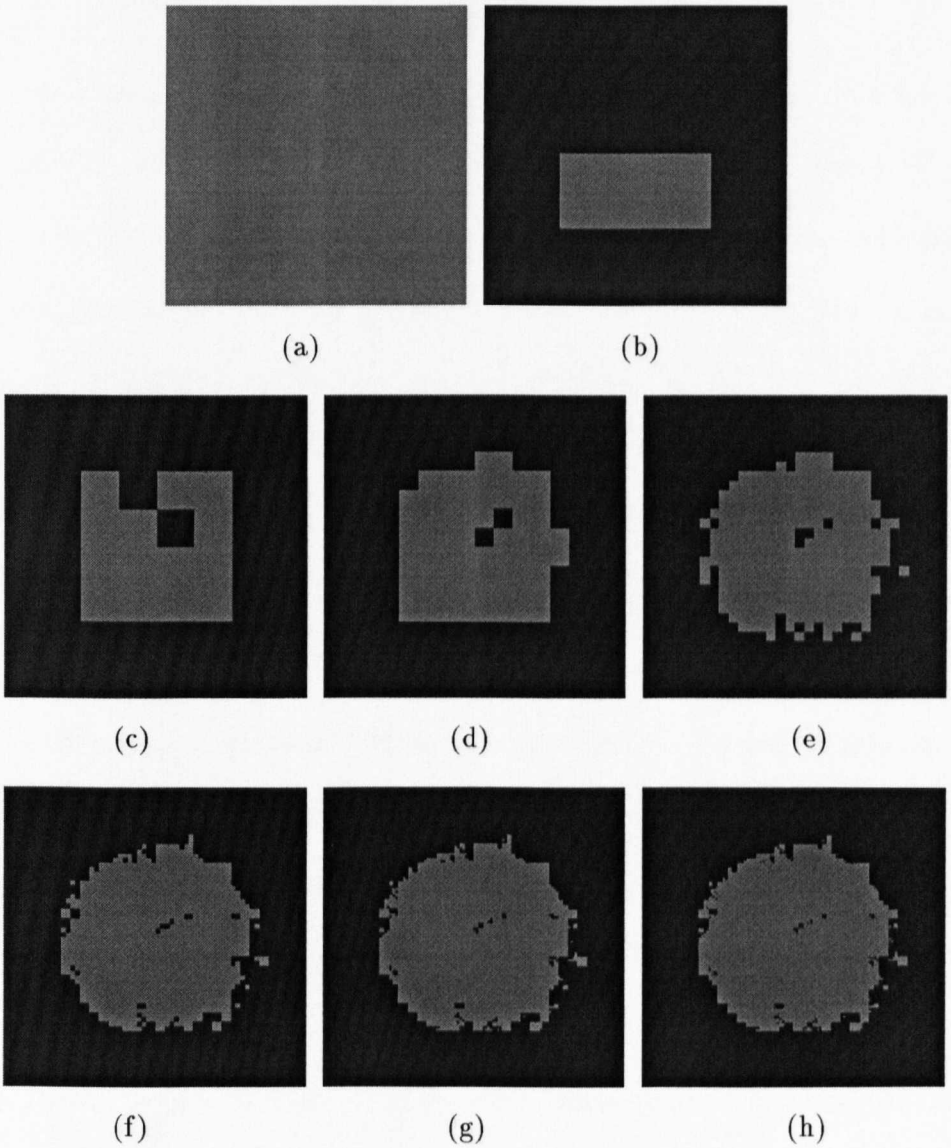


Figure 5.8: The intermediate segmentation maps of Figure 5.6(a) at different levels using a *margin_ratio* of 0.37. From (a)-(h): level 1-8.

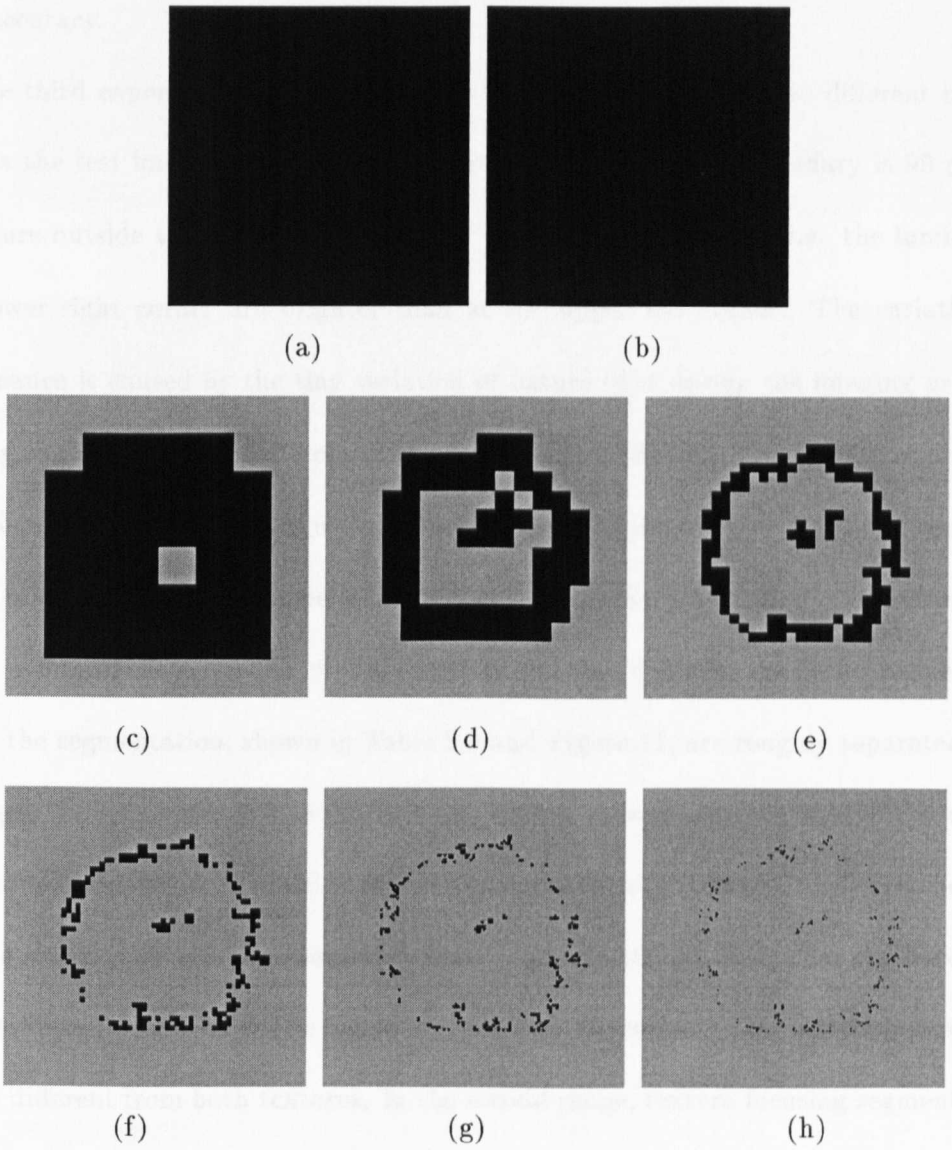


Figure 5.9: The intermediate fixed regions (shown as grey) at different levels using a *margin_ratio* of 0.37. From (a)-(h): level 1-8.

texture region is fixed in the earlier stage (i.e. coarser resolutions). The estimations of the textural contents in the border areas are prone to inaccuracy if coarse resolutions are used. Therefore, these areas are fixed at the finer resolutions to locate the boundary with greater accuracy.

In the third experiment, a 256x256 image which is composed of two different metals is used as the test image (Figure 5.10(a)) where the radius of the boundary is 90 pixels. The texture outside the boundary contains a variation in luminance, i.e. the luminance at the lower right corner are brighter than at the upper left corner. The variation of the luminance is caused by the tiny variation of nature light during the imaging process. According to the texture model proposed by Francos *et.al.* [29], the variations in luminance are the generalised evanescent components which can be detected by a 1-D filter tuned to a certain orientation. Since texture focusing does not consider orientation information, the variation in luminance is not interpreted correctly and thus degrades the performance. The PCS's of the segmentation, shown in Table 5.3 and Figure 11, are roughly separated into four ranges: $margin\_ratio \leq 0.18$ (PCS < 50); $0.19 \leq margin\_ratio \leq 0.30$ (PCS ≈ 65); $0.31 \leq margin\_ratio \leq 0.37$ (PCS > 90) and $margin\_ratio \geq 0.38$ (PCS=0). In the first range, the image is divided into several regions (Figure 5.10(b)). Note that the border of the two textures is classified as one region because it is the transient area where the textural feature is different from both textures. In the second range, texture focusing segments the region outside the boundary into two classes due to the variations in luminance (see Figure 5.10(c) and (d)). Comparing Figure 5.7 with Figure 5.11, most of the *margin_ratios* in this range result in a satisfactory segmentation in the first experiment. In the third range, the PCS's reach as high as 93 (Figure 5.10(e) and (f)). In the final range, texture focusing cannot distinguish the two textures in the image, and the segmentation maps are identical to Figure 5.6(f). This experiment shows that the variations in luminance degrades the

performance of texture focusing. However, if the *margin_ratios* is chosen properly, a good segmentation can still be achieved.

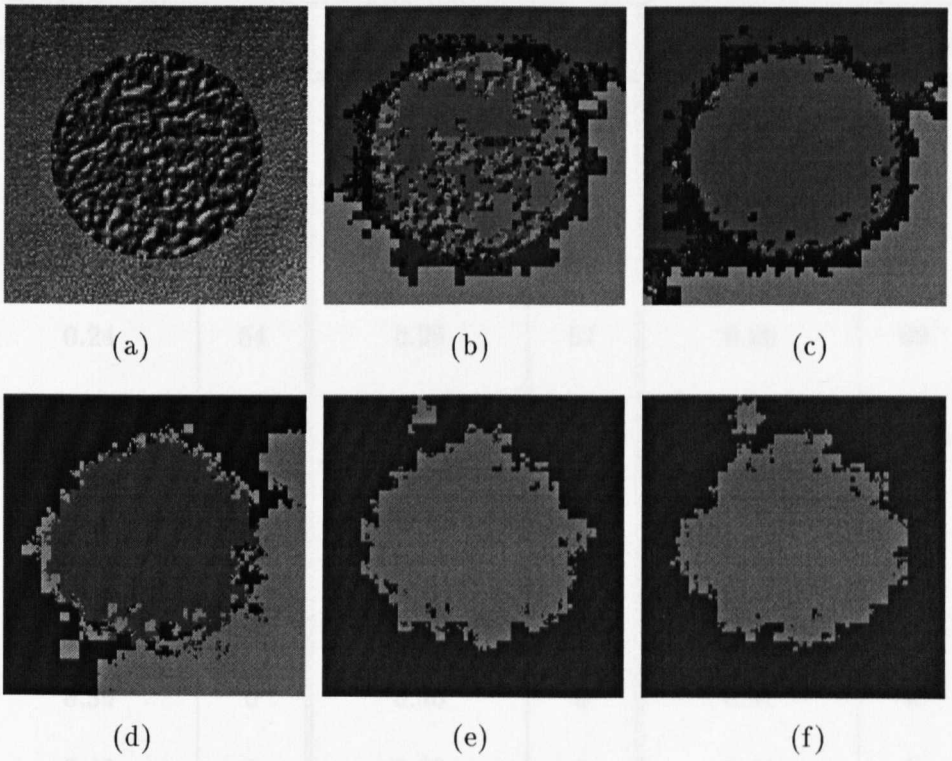


Figure 5.10: (a) An image with textures of two different metals. The segmentation map of (a) using a : (b) *margin_ratio* of 0.17 (PCS=39); (c) *margin_ratio* = 0.20 (PCS=59); (d) *margin_ratio* = 0.27 (PCS=69); (e) *margin_ratio* = 0.31 (PCS=93); and (f) *margin_ratio* = 0.35 (PCS=93).

5.5 Summary

This chapter presents texture focusing, an efficient multiresolution image segmentation scheme which is computationally modest. Texture focusing comprises the concepts of the spatio-featural mutual focusing and the split-and-fix process. The spatio-featural mutual focusing achieves high resolutions in both the spatial and the featural domains for texture segmentation. The split-and-fix process employs the contextual information to indicate the

Table 5.3: PCS's of segmentation maps of Figure 5.10(a) using various *margin_ratios*.

margin_ratios	PCS's	*margin_ratios*	PCS's	*margin_ratios*	PCS's
0.15	33	0.16	33	0.17	39
0.18	41	0.19	59	0.20	59
0.21	56	0.22	63	0.23	63
0.24	64	0.25	67	0.26	69
0.27	69	0.28	66	0.29	66
0.30	70	0.31	93	0.32	93
0.33	93	0.34	93	0.35	93
0.36	92	0.37	91	0.38	0
0.39	0	0.40	0	0.41	0
0.42	0	0.43	0	0.44	0

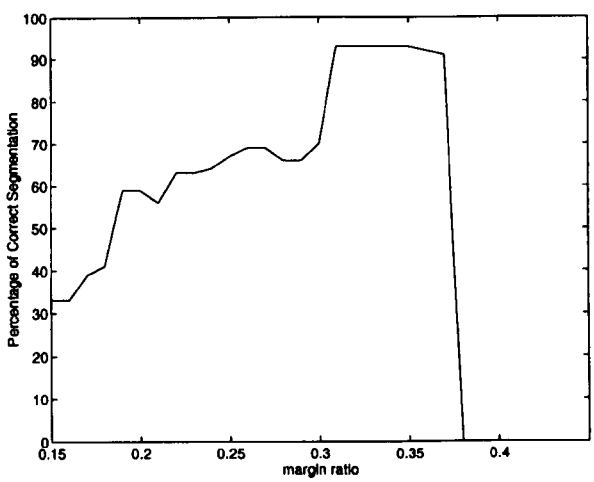


Figure 5.11: The percentages of the correct segmentation (PCS's) of the segmentation results of Figure 5.10(a) using various *margin_ratios*.

adequate resolution in every region of the segmentation map. LTVP, a concept motivated by the Metropolis probability of the simulated annealing, is also employed in texture focusing to adjust the probability of a change of state according to the image resolution. Texture focusing results in a multiresolution segmentation map, where the central regions of homogeneous textures are represented using coarse resolutions so as to achieve a better estimation of the textural content, and the border region of a texture is represented using fine resolutions so as to achieve a better estimation of the boundary between textures.

A measurement of PCS is proposed for the quantitative evaluation of the performance of texture focusing, where a higher PCS indicates a better segmentation. The experimental results show that a segmentation with PCS higher than 90 can be obtained using a wide range of *margin_ratios*. The highest PCS obtained is 97. The variations in luminance of the textural image influences the performance. However, a proper selection of the *margin_ratio* results in a satisfactory segmentation map.

Chapter 6

Motion Field Segmentation

6.1 Uncertainty, Ill-posedness and Ill-conditioness

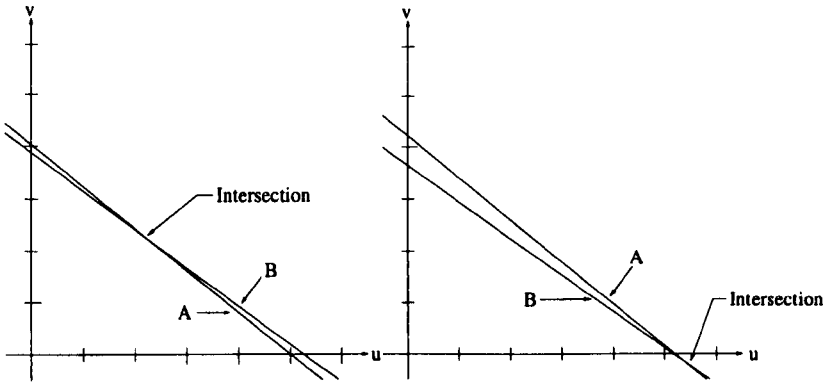
The tracking of moving objects through a sequence of images is one of the important tasks in computer vision. It is a real-time task in which motion is an important cue for the detection of targets. Meyer and Bouthemy argue that the process of object tracking should comprise two stages: the detection of a moving target; and the pursuit of the target [69]. The first stage aims to detect the moving objects from the background using an optical flow, a dense motion field extracted from an image sequence (see Section 1.3). A parametric motion model of the target is then derived which serves as the basis for the second stage of the tracking. There are other methods which depend on pre-defined object models (e.g. [8]) or certain image features (e.g. [88]). The Meyer and Bouthemy's approach is adopted in this thesis because it requires the least amount of *a priori* information.

As indicated by equation (1.1), the optical flow constraint equation (OFC) is derived under the assumption that the image grey-level of a moving point is stationary with respect to time. It is an ill-posed task to derive the motion field (a field of 2-D velocity vectors) directly from OFC's, because two variables of u and v are determined using

a single constraint equation, i.e. the aperture problem. To be more precise, only the velocity component perpendicular to the isophote curve of the image is determined in OFC's (see [4] and Section 6.3.1). In addition, the coefficients of OFC (i.e. three partial derivatives of $\frac{\partial S}{\partial x}$, $\frac{\partial S}{\partial y}$ and $\frac{\partial S}{\partial t}$) are very sensitive to noise. Furthermore, OFC is derived under the assumption that $S(x, y, t)$ is continuous, which will fail at discontinuities in real images. Three approaches have been proposed to overcome the above three limitations: the multi-constraint approach, the regularisation approach and the multi-point approach [26]. The multi-constraint methods involve a global smoothing stage to suppress noise [26]. The regularisation approach, which smooths the motion field, normally involves the calculus of variations or the labelling of Markov random fields (MRF) using the maximum *a posteriori* (MAP) method. For example, Meyer and Bouthemy [69] employ the MRF-MAP method proposed by Bouthemy and Francois [10] as their first stage of tracking. Both the calculus of variations and the MRF-MAP method are computationally expensive, thus their implementation for real-time tracking is unrealistic.

The multi-point approach assumes that the optical flow of adjacent image pixels are almost identical. Thus, the OFC's of pixels within an operator window are used to derive an over-determined set of equations, where the solution (obtained by the least-square method) represents the group velocity of these pixels. Note that a window which spans across object boundaries causes an erroneous estimation of the velocity due to the discontinuity in the motion field. Nesi *et al.* thus propose a complex method which estimates the velocity (u, v) from the intersection points of the OFC's in the $u - v$ solution plane [76]. However, the ill-conditioned nature of the over-determined sets of OFC's makes these intersection points unreliable [51]. The ill-conditioness means a small perturbation in the input signal results in a large variation of the output (see Section 3.3.2). For example, Figure 6.1(a) shows two OFC's $0.2u + 0.25v - 1 = 0$ and $0.19u + 0.26v - 1 = 0$, which

intersect at the position $(2.2222, 2.2222)$ of the $u - v$ plane. If $\frac{\partial E}{\partial t}$ is slightly perturbed (due to noise) as in Figure 6.1(b) ($0.2u + 0.25v - 1.03 = 0$ and $0.19u + 0.26v - 0.97 = 0$), then the intersection changes significantly to $(5.6222, -0.3778)$. Thus, the estimation by intersection is not reliable.



(a) (b)

Figure 6.1: (a) Two OFC's ($0.2u + 0.25v - 1 = 0$ and $0.19u + 0.26v - 1 = 0$) in the $u - v$ plane. (b) The OFC's with a slightly perturbed coefficients ($0.2u + 0.25v - 1.03 = 0$ and $0.19u + 0.26v - 0.97 = 0$).

All the above methods are single-resolution methods. However, the size of the window influences the estimation of the motion field. OFC's of a small window (i.e. high spatial resolution) tend to result in an erroneous solution due to the ill-conditioness. If a large window is used (i.e. low spatial resolution), then the estimated optical flow is an averaged value within the window, which is less likely to be influenced by noise but is more likely to contain multiple objects. This is the uncertainty problem which can be circumvented by the use of multiresolution approaches (see Sections 1.4 and 5.2). In addition, the use of the OFC's of the image pixels of the same physical object achieves a better estimation of the motion field. However, a dilemma emerges that object boundaries are not known until the motion field has been adequately estimated. This is the dilemma of segmentation which

also occurs in the task of texture segmentation (see Section 5.3.2). The multiresolution clustering method of Texture Focusing proposed in Chapter 5 has already solved the above two dilemmas. It is therefore extended to achieve the motion field segmentation.

6.2 The Design of a Motion Field Segmentation Scheme

6.2.1 Optical Flow Pyramid

The proposed motion field segmentation scheme is based on the multi-point method and the multiresolution clustering method of Texture Focusing. This is an attempt to generalise the multiresolution clustering method as an image segmentation framework. In the proposed method, an optical flow pyramid is generated, which is used by the subsequent multiresolution clustering process in producing a segmentation map. First, the spatial Gaussian filtering with various standard deviations (σ) is applied on two temporally adjacent frames to construct two Gaussian pyramids [13]. This is to reduce noise as well as the temporal aliasing caused by large movements of objects [4]. Note that the Gaussian pyramid is a quad-tree image structure which provides a natural over-determined set of OFC's, i.e. the OFC's of the four children nodes determine the group velocity of the image pixels associated with the parent node (Figure 6.2). Second, the Gaussian pyramids of the current frame $S_L(x, y, t)$ and the previous frame $S_L(x, y, t-1)$ are used to determine the coefficients of the OFC of each node, i.e. the three derivatives, via the finite difference method (see Section 2.1):

$$\frac{\partial S_L}{\partial t} = S_L(x, y, t) - S_L(x, y, t-1)\,,$$

$$\frac{\partial S_L}{\partial x} = \frac{S_L(x+1, y, t) - S_L(x-1, y, t)}{2 \times window\_size_L}\,,$$

$$\frac{\partial S_L}{\partial y} = \frac{S_L(x, y+1, t) - S_L(x, y-1, t)}{2 \times window\_size_L}\,,$$

where L indicates the level of the quad-tree, and $window\_size_L$ is the width of the block-shaped window at level L of the quad-tree.

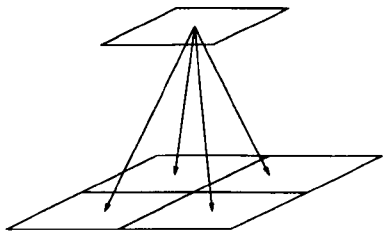


Figure 6.2: The parent-children relationship in a quad-tree: a unit for an over-determined set of OFC's

Third, the velocity (u, v) of a node in the optical flow pyramid is determined using the set of OFC's of its children nodes. The solution of the OFC's are achieved via the least-square method. Define $\vec{u} \equiv (u, v)$, and denote A and $\vec{b}$ as the coefficient matrx and coefficient vector of the over-determined set of OFC's, i.e. $A\vec{u} = \vec{b}$. The solution of $\vec{u}$ is obtained by the minimisation of the least-square error E (i.e. the sum of the square error):

$$E = (A\vec{u} - \vec{b})^T(A\vec{u} - \vec{b}). \tag{6.1}$$

The minimum of E occurs when

$$(A^T A)\vec{u} = A^T\vec{b}.$$

If $det\left[A^T A\right] \gg 0$, then a unique solution of $\vec{u}$ is determined as $(A^T A)^{-1}A^T\vec{b}$. If $det\left[A^T A\right] \approx 0$, then a unique solution is still achieved which however is susceptible to noise due to the ill-conditioness. Otherwise, $\vec{u}$ is undetermined when $det\left[A^T A\right] = 0$. Define

$$A \equiv \left[\begin{array}{cc} \vec{p} & \vec{q} \end{array}\right],$$

where $\vec{p}$ and $\vec{q}$ are 4×1 vectors. Thus

$$det\left[A^T A\right] = 0$$

$$\Longrightarrow \quad det(\begin{bmatrix} \vec{p} \\ \vec{q} \end{bmatrix} \begin{bmatrix} \vec{p} & \vec{q} \end{bmatrix}) = 0$$

$$\Longrightarrow \quad det \begin{bmatrix} \vec{p}\vec{p} & \vec{p}\vec{q} \\ \vec{q}\vec{p} & \vec{q}\vec{q} \end{bmatrix} = 0$$

$$\Longrightarrow \quad \|\vec{p}\|^2\|\vec{q}\|^2 = \|\vec{p}\|^2\|\vec{q}\|^2 \cos^2\theta,$$

where $\|\cdot\|$ denotes the l_2 norm, and θ is the angle between $\vec{p}$ and $\vec{q}$. The solutions of the above equation are either $\vec{p} = 0$, $\vec{q} = 0$ or $\theta = 0$, which are the direct consequences of the aperture problem. If $\vec{p} = 0$, then u is undetermined. If $\vec{q} = 0$, then v is undetermined. If $\theta = 0$, then vectors $\vec{p}$ and $\vec{q}$ have the same orientation, thus the solution of $\vec{u}$ is a line on the $u - v$ plane, which is composed of a determined vector $\vec{u_o}$ and a undetermined vector $\vec{u_1}$ (see Figure 6.3), where $\vec{u_o}(u_o, v_o)$ is determined by the following equation:

$$u_o = \frac{\vec{p} \cdot \vec{b}}{\|\vec{p}\|^2 + \|\vec{q}\|^2} \qquad v_o = \frac{\vec{q} \cdot \vec{b}}{\|\vec{p}\|^2 + \|\vec{q}\|^2}.$$

In the motion field segmentation scheme, the undetermined vector $\vec{u_1}$ is assumed to be a zero vector because the velocity component along the isophote orientation cannot be detected using the OFC's (i.e. the aperture problem).

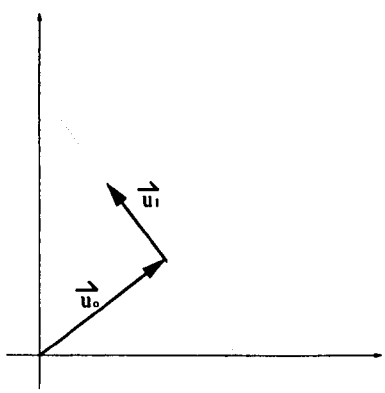


Figure 6.3: The $u - v$ plane with a determined vector $\vec{u_o}$ and a undetermined vector $\vec{u_1}$

6.2.2 Multiresolution clustering

As presented in Section 5.3, the multiresolution clustering method is a coarse-to-fine process operating on a quad-tree. In motion field segmentation, a moving object is assumed to generate a homogeneous motion field in the image, thus the pixels with similar value of u and v are clustered together, i.e. the $u - v$ solution plane corresponds to the feature space of the clustering. The multiresolution clustering comprises the spatio-featural mutual focusing and the split-and-fix process. The basic idea behind the spatio-featural mutual focusing is that a good spatial estimation of the object boundary results in a good estimation of the motion field, and vice versa. Thus, they are useful in improving the estimation of each other in a multiresolution process. The coarse-to-fine process is employed to locate the object boundaries with increasing resolutions. Feature focusing is employed in conjunction with the spatial focusing to indicate an accurate cluster centre (i.e. u and v) in the feature space.

The least-square error of each node is a by-product (determined according to equation (6.1)) in the construction of the optical flow pyramid. These errors are used to indicate an adequate resolution for each region. If an error of a node is smaller than a pre-defined threshold (referred to as the *fix_threshold*), then the motion field within the window is assumed to be reasonably homogeneous, thus the spatial resolution of this node is not increased any further. This is referred to as the split-and-fix process, where the adequate resolutions in all the regions of the segmentation map are indicated by the homogeneity of the motion field. This results in a multiresolution segmentation map, where the central region of homogeneous motion fields are represented using coarse resolutions so as to achieve a better estimation of $\vec{u}$ (in the sense of well-conditioness), and the border regions are represented using fine resolutions so as to achieve a better estimation of the boundaries

between the homogeneous motion fields (see Figure 6.4).

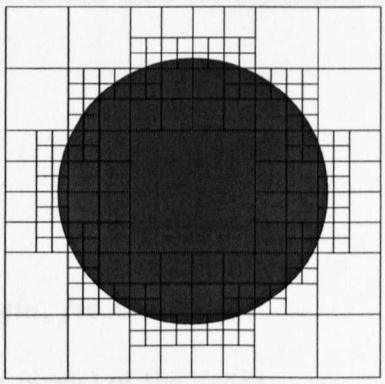


Figure 6.4: A multiresolution representation of an image. The two motion fields are depicted as grey and white. The central regions of homogeneous motion fields are represented using large windows, while the border regions of textures are represented using small windows.

6.2.3 The Algorithm

In the implementation, two quad-trees are used respectively to store the optical flow pyramid (denoted as feature (L, node)) and the class (denoted as class (L, node)) in a multiresolution data structure. A dynamic chain is used to store the class information (e.g. *cluster_centres*) of all the classes. The data structures and pseudo codes are as follows:

Definitions of data structures

quad-tree: feature (L, node)={*u, v, error*};

/* the optical flow and the least-square error of a node at level L */

quad-tree: class (L, node)={*class_pointer, fixed*};

/* the class of a node at level L */

dynamic chain: class_table (*class_pointer*)={*cluster_centre, class_number*};

/* a table of cluster centres and their corresponding number of nodes which belong to the class */

Pseudo Codes

begin

Input (Image, *basis_margin*, *fix_threshold*);

Assign w = the width (in pixels) of the image;

Assign *total_level* = log_2 (w);

Assign margin = *basis_margin*;

Construct level 3 of the Gaussian pyramids ($S_3(x, y, t)$ and $S_3(x, y, t-1)$);

Determine feature (2, node) using $S_3(x, y, t)$ and $S_3(x, y, t-1)$;

Create new_class = { the average of feature (2, node), 16 };

Append new_class to class_table;

For every node at level 2 **do**

Determine p1 = feature (2, node), p2 = class_table(class (L, node));

Determine *dist* = distance (p1, p2);

Determine $LTVP(assign)$ using *dist* and *margin*;

If ($random\_no \geq LTVP(assign)$)

begin

Create new_class = { feature (L, node), 1 };

Append new_class to class_table;

Assign class (L, node) = new_class;

end

For L=3 to (*total_level* − 1) **do**

begin

/* split */

Propagate all the class (L-1, parent) to their children nodes' class (L, children);

For every node at level L **do**

begin

Construct $S_{L+1}(x, y, t)$ and $S_{L+1}(x, y, t-1)$;

Determine feature (L, node) using $S_{L+1}(x, y, t)$ and $S_{L+1}(x, y, t-1)$;

If (class (L, node) $\neq$ fixed)

begin

Determine p1 = feature (L, node), p2 = class_table (class (L, node));

Determine *dist* = distance (p1, p2);

Determine $LTVP(assign)$ using *dist* and *margin*;

If $(random\_no \geq LTVP(assign))$

begin /* search for a new class */

For every class in the class_table **do**

Determine *dist* = distance (*cluster_centre*, feature (L, node));

Determine the closest_class such that *dist* is minimised;

If L=*total_level* − 2 or *total_level* − 1

Assign class (L,node) = closest_class;

else

begin

Determine $LTVP(assign)$ using *dist* of the closest_class and *margin*;

If $(randon\_no \leq LTVP(assign))$

Assign class (L,node) = closest_class;

else

Create new_class = { feature (L, node), 1 };

Append new_class to class_table;

Assign class (L, node) = new_class;

end

end

end

end

Assign margin = margin + *basis_margin*;

/* fix */

For every node at level L **do**

begin

If *error* of feature (L, node) $\leq$ *fix_threshold*

class (L,node)=fixed; /* update un-fixed pixels to fixed */

end

For every class in the class_table **do** /* feature focusing */

begin

If (*class_number*=0)

Eliminate the current class from the class_table;

else

Determine *cluster_centre* using { feature (L, node) | class (L, node) = current class };

end

end

Produce Segmentation Map according to class (total_level, node);

Output (Segmentation Map)

end

6.2.4 Comparison of Texture and Motion Field Segmentation

The proposed motion field segmentation scheme is different from the Texture Focusing scheme in the following aspects:

1. In the motion field segmentation scheme, the Gaussian pyramid is employed to reduce the temporal-aliasing and noise. In Texture Focusing, the Gaussian pyramid is not used.

2. In the motion field segmentation scheme, the size of the window is a trade-off between well-conditioness and the spatial resolution. In Texture Focusing, the size of the window is a trade-off between the spatial and feature resolutions.

3. In the motion field segmentation scheme, the feature space corresponds to the 2-D $u - v$ solution plane. In Texture Focusing, the 3-D feature space corresponds to the statistics of the texture content.

4. In the motion field segmentation scheme, the least-square error is used to determine the fixing of a node. In Texture Focusing, the contextual information is used to determine the fixing of a node.

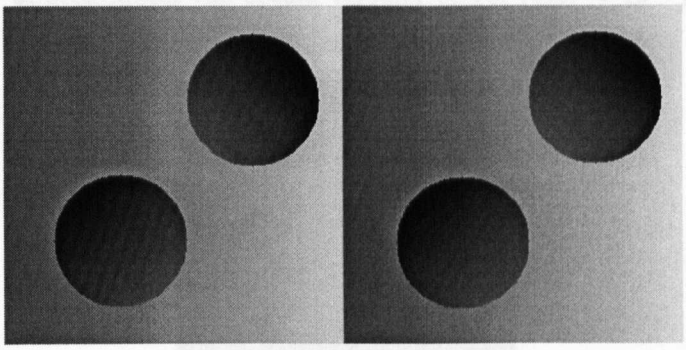
5. The motion field segmentation scheme starts from the Level 2 of the quad-tree, where the average of the optical flows determined at Level 2 is used as the first cluster centre. In Texture Focusing, the textural feature of the root node of the quad-tree is used as the first cluster centre.

6. In the motion field segmentation scheme, the mechanism to create a new class is switched off at the last two levels. In Texture Focusing, this mechanism remains

active at all levels.

6.3 Performance Evaluations

Two synthetic images with 256×256 pixels (Figures 6.5(a) and (b)) are designed as two temporally adjacent frames for the evaluation of the proposed motion field segmentation scheme. The u direction is defined as the vertical downward direction and the v direction is the horizontal rightward direction. The luminance of the background in the two frames varies linearly, where the isophote lines (the vertical lines) are shifted four pixels to the right ($v = 4$ pixels/frame). This is to simulate a pan of the camera to the left. Two circular disks, each of a radius of 50 pixels, are used to simulate two moving objects with different velocities. The upper-right disk moves 2 pixels downward ($u = 2$ pixels/frame) and 2 pixels to the left ($v = -2$ pixels/frame). The lower-left disk moves 2 pixels upward ($u = -2$ pixels/frame) and 2 pixels to the left ($v = -2$ pixels/frame). These movements are chosen to cause a temporal aliasing situation (i.e. the velocity of the motion is greater than 1 pixel/frame [4]).



(a) (b)

Figure 6.5: (a) An image frame in a sequence of synthetic images and (b) its previous frame.

Figure 6.6 shows the segmentation result produced by the proposed scheme using the

parameters of $basis\_margin$ = 0.2 pixel/frame and $fix\_threshold$ = 1. There are 33 optical flows (i.e. the cluster centres in the feature space) detected in this image, where different optical flows are shown as different grey levels in the segmentation map. Table 6.1 shows the actual velocities $\vec{u}_{actual}$, the estimated velocities $\vec{u}_{estimated}$, the error ratios and the numbers of pixels of three major regions with distinct group velocities, which correspond to the two disks and the background. The error ratio is defined as

$$\frac{\|\vec{u}_{actual} - \vec{u}_{estimated}\|}{\|\vec{u}_{actual}\|}.$$

Both of the two disks are classified as homogeneous motion fields, and the estimated velocities approximates the actual velocity (error ratio $\approx 9 \times 10^{-4}$). The number of pixels of these two regions are 7272 and 7276 respectively, which approximately corresponds to circles with a radius of 48 pixles. The estimated circle is smaller than the actual radius of 50 pixels. This is because the assumption of $S(x, y, t)$ being continuous is violated in the border regions of the disks (see Section 6.1) where a complex motion field is generated. The majority of the background is classified as a homogeneous motion field where the estimated velocity approximates the actual velocity (error ratio $\approx 10^{-4}$).

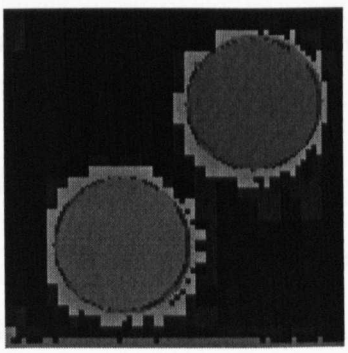


Figure 6.6: The segmentation map of the motion field ($basis\_margin$ = 0.2 pixel/frame and $fix\_threshold$ = 1).

Table 6.1: The actual velocities, the estimated velocities, the error ratios and the numbers of pixels of the three major motion fields in Figure 6.5.

		Actual velocity (pixel/frame)	Estimated velocity (pixel/frame)	Error ratio	No. pixels
Background	u	0.0	−0.000090	1.5×10^4	31076
	v	4.0	3.999393		
Upper-right	u	2.0	2.000460	9.7×10^4	7272
Disk	v	−2.0	−1.997283		
Lower-left	u	−2.0	−1.998474	8.9×10^4	7276
Disk	v	−2.0	−1.996765		

There are 19912 pixels (i.e. 30.38 per cent of the image) which are not classified as the three major classes. Most of these mis-classified regions lie on the border of the disks due to the discontinuity of the grey levels and the motion field. The result of the segmentation is less satisfactory where the boundary of the motion fields need to be accurately detected, however, it provides a quick estimation of the moving objects with their distinct velocities when the camera is moving, which is a basic requirement of the first stage of object tracking.

There are still difficulties for the implementation of the motion field segmentation scheme on real images. This is because the assumption that a moving object corresponds to a homogeneous optical flow field is not necessarily true in real images (see Section 6.2.2).

6.4 Summary

The motion field segmentation scheme proposed in this chapter is an attempt to generalise the multiresolution clustering method as an image segmentation framework. The motion field segmentation using the OFC's is chosen to test the applicability of the multiresolution clustering method on difficult tasks. The result of the scheme detects distinct moving objects with their velocities, thus fulfilling the basic requirement of the first stage of object tracking.

Chapter 7

Conclusions

7.1 Achievements in this Thesis

This thesis presents the investigations of two different tasks in the discipline of computer vision: edge detection (Chapters 2-4) and image segmentation, which includes texture segmentation (Chapter 5) and motion field segmentation (Chapter 6). These investigations result in a general observation that the uncertainty principle is a natural limitation in image analyses, thus adaptive multiscale/multiresolution methods are required to circumvent the uncertainty.

Chapter 2 presents a review of edge detectors to illustrate the important concepts for edge detection, particularly in the comparison between the isotropic differentiator of ∇^2 and the directional differentiator of $\frac{\partial^2}{\partial n^2}$. Even though the theoretical analysis shows that the $\frac{\partial^2}{\partial n^2}$ differentiator (as in the Haralick scheme) is preferable to ∇^2 (as in the LoG scheme), the standard deviation of Gaussian in the LoG scheme provides a natural scale parameter for the development of the scale-space theory [108].

Chapter 3 examines the concept of regularisation, which is an important technique for formulating a well-posed task. Section 3.3.1 shows that the Cubic B-spline fitting

transforms a regularisation formula into a quadratic energy function. The ill-conditioness of the Regularised Cubic B-Spline fitting is also examined in Section 3.3.2, where the Regularised Cubic B-Spline fitting is modified to achieve a better fitting. A roof edge detector is also presented which employs this modified Regularised Cubic B-Spline fitting.

The Bounded Diffusion theory, a logical consequence of the uncertainty principle, is presented in Section 4.2.1. The α scale space shows the diffusive/convergent edge behaviour of Bounded Diffusion (Section 4.2.2). The multiscale edge detector of MRCBS is based on the Bounded Diffusion, where the size of the operator kernel is fixed to preclude the irrelevant information from the smoothing process. In addition, the finest scale is adaptively adjusted according to the local noise level. Furthermore, a series of thresholds with the same thresholding capability is used in the corresponding scales to prevent noise clusters. A thorough evaluation of the performance of four edge detectors, i.e. MRCBS, the Edge Focusing scheme, the Chen/Yang edge detector and the Haralick scheme (Section 4.4) shows the superiority of MRCBS over the other three schemes.

An adaptive multiresolution clustering scheme of Texture Focusing is presented in Chapter 5. Texture focusing comprises the concepts of the spatio-featural mutual focusing and the split-and-fix process. The spatio-featural mutual focusing achieves high resolutions in both the spatial and the featural domains for texture segmentation. The split-and-fix process employs the contextual information to indicate the adequate resolution in every region of the segmentation map. Texture focusing results in a multiresolution segmentation map, where the central regions of homogeneous textures are represented using coarse resolutions so as to achieve a better estimation of the textural content, and the border regions of textures are represented using fine resolutions so as to achieve a better estimation of the boundary between textures.

The success of Texture Focusing in the task of texture segmentation motivates the

use of the multiresolution clustering method together with the multi-point optical flow method to tackle the problem of motion field segmentation. This is an attempt to study the potential of the multiresolution clustering method as a generalised image segmentation framework. However, the performance of the segmentation via the optical flow constraint equations (OFC's) is highly constrained by the aperture problem (Section 6.1) and therefore, subsequent investigation is required to improve the performance of this scheme to achieve an accurate motion field segmentation.

There are two major theoretical achievements in this thesis: the α scale space for a multiscale edge detector, and the multiresolution clustering method for texture segmentation and motion field segmentation. Both of these two achievements are derived from a general observation of the uncertainty principle. The α scale space is proposed because an edge is a local property, thus the local scale factor of α controls the degree of smoothing according to the local noise level. On the contrary, textures and motion fields are regional properties, thus a multiresolution representation of the image (see Figure 5.2) is proposed so that the regional property is estimated using large windows, whereas the location of the boundaries is estimated using small windows.

7.2 Toward a Generalised Theory of Adaptivity

Wilson and Spann argued that image processing algorithms can be categorised as two groups: those derived heuristically and those derived rigorously within the framework of signal processing theory [105]. The former are either difficult to be extended to new applications or difficult to be compared objectively with each other. The latter tend to employ assumptions which are inadequate for real images [105]. In this regard, they advocate the research on a generalised theory which provides a solid theoretical background

for the unification of image processing algorithms [105].

The research presented in this thesis is a response to the above advocacy. Since a generalised theory is meaningless unless it is based on the observation of real situations, two low-level computer vision tasks of edge detection and image segmentation are selected as the subjects of the investigation. During the investigation, the importance of the uncertainty principle emerges, which is a candidate as the central issue of a generalised theory. An adequate understanding of the uncertainty distinguishes the difference between the local processing of edge detection and the regional processing of image segmentation. A multiresolution scheme such as Texure Focusing makes the most of the spatial-featural information by using different resolutions in different portions of an image.

Adaptivity is the common feature for both the multiscale edge detector and the multiresolution clustering scheme presented in this thesis. These two schemes embody a common mechanism, which comprises a coarse-to-fine procedure with a decrement of the scale, and a measure (i.e. EHF for multiscale edge detection, the contextual information for Texture Focusing and the least-square error for the motion field segmentation) indicates an adequate scale/resolution for each region of the image. This common mechanism fully exploits the adaptivity in a multiscale/multiresolution scheme, and thus provides a solution for the uncertainty inherent in the image analysis.

Therefore, this thesis starts from the general observation of the uncertainty, and finishes with the general solution of the adaptivity which provides a prototype of a generalised theory for image analysis. However, this is still far from a form of a rigorous theory derived by mathematics. First, there are unsolved problems when the multiresolution clustering scheme is extended for the motion field segmentation. Second, most of the algorithms are heuristically derived. The investigation on a generalised theory is thus still in progress, which is the long term goal of this research.

Appendix A

List of Publications

A.1 Journal papers

- Kung-Hao Liang, Tardi Tjahjadi, Yee-Hong Yang, "Roof Edge Detection using Regularized Cubic B-Spline Fitting," Pattern Recognition, vol.30(5), pp.719-728, 1997.
- Kung-Hao Liang, Tardi Tjahjadi, and Yee-Hong Yang, "Bounded Diffusion for Multiscale Edge Detection using Regularized Cubic B-Spline Fitting," submitted to IEEE Transactions on System, Man and Cybernetics, in September 1996, awaiting review.
- Kung-Hao Liang, Tardi Tjahjadi, and Yee-Hong Yang, "Texture Focusing: A Multiresolution Approach for Segmentation," submitted to Pattern Recognition in May 1997, awaiting review.

A.2 Conference papers

- Kung-Hao Liang, Tardi Tjahjadi, Yee-Hong Yang, "A Regularized Multiscale Edge Detection Scheme using Cubic B-Spline," in Proc. UK Symposium on Applications of Time-Frequency and Time-Scale Methods (TFTS'95), IEEE Signal Processing

Chapter (UKRI Section), Coventry, UK, pp.58-65, 1995.

- Kung-Hao Liang, Tardi Tjahjadi and Yee-Hong Yang, "Multiscale Texture Segmentation based on Image Spectrum," in Proc. IEEE Nordic Signal Processing Symposium (NORSIG'96), Helsinki, Finland, pp. 239-242, 1996.

- Kung-Hao Liang and Tardi Tjahjadi, "Spatio-Temporal Filtering for Moving Objects Tracking," in Proc. 2nd IEEE UK Symposium on the Applications of Time-Frequency and Time-Scale Methods (TFTS'97), Coventry, U.K., pp. 53-56, 1997.

Appendix B

The codes of the roof edge detector and for the computation of FCR/ATR.

```
/* ************************************************************************ */
/* Name of the programme: roof.c                                            */
/*                                                                          */
/* Function: Roof edge detector using Regularised Cubic B-Spline Fitting    */
/* ************************************************************************ */
/* To Compile this source code: cc -o roof  roof.c -lm                      */
/*                                                                          */
/* To run the programme:  roof (then initiate the user-friendly input query)*/
/*                                                                          */
/* Projection width: the number of pixels for the equal-weighted averaging. */
/* (Recommanded value=1)                                                    */
/* No. of projected data: the number of pixels for the local operation.     */
/* (Recommanded value=7)                                                    */
/* Regularizing factor: Recommanded value=0.1                               */
/* ************************************************************************ */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* ******************************************** */
/* MaxWidth is the maximum number of Pjsize.    */
/* NumPoint is the maximum number of M.         */
/* ImagSize is the maximum number of image width. */
/* BUFSIZE is the default buffer size.          */
/* WHITEVALUE represents the white grey level.  */
/* BLACKVALUE represents the black grey level.  */
/* ******************************************** */

#define MaxWidth   5
#define NumPoint   10
#define ImagSize   512
#define BUFSIZE    512
#define WHITEVALUE 255
#define TRUE       0
#define FALSE      255
#define BLACKVALUE 0
#define Tmax       256
#define PI         3.14159263536

int   ras_magic, ras_height, ras_type,  ras_maptype;
int   ras_width, ras_length, ras_depth, ras_maplength;

/* ************************************************** */
/* V : Number of B-splines for curve fitting.         */
/* Pjsize : The width of the projection and the gradient.*/
/* MaskWidth : The width of the submasks f(i).        */
/* threshold : The value is used for determining edgels. */
/* B : The regularizing factor.                       */
/* Aimag : The angle of Gradient.                     */
/* Mimag : The magitude of Gradient.                  */
/* ************************************************** */

int   threshold;
int   V,M,Pjsize,MaskWidth;
float Zero;
char  *IPfile,*OPfile;
char  IPname[15],OPname[15];
float B,Aimag,Mimag;
float S2,H2,H3L,H3R,V2,V3L,V3R,S1,H1R,H1L,V1R,V1L;
int   Timage[ImagSize][ImagSize],Map[ImagSize][ImagSize];
float f[NumPoint+2],d[NumPoint+2][NumPoint+2];
float A0[NumPoint][NumPoint+2],A2[NumPoint][NumPoint+2];
float alpha[NumPoint+2][NumPoint+2],beta[NumPoint+2][NumPoint+2];
float mask[NumPoint+MaxWidth+2][NumPoint+MaxWidth+2];
```



```
/* ***************************** */
/* MAIN PROGRAM OF EDGE DETECTION */
/* ***************************** */
main()
{

IPproc();
printf("\n\n\Initialize");
Initialize();
printf("\n\n\fitting");
Fitting();
printf("\n\nOutput");
OPproc();
}

/* ********************** */
/* INPUT ALL REQUIREMENTS */
/* ********************** */
IPproc()
{
int    ascii;
char   *temp="\0";

printf("Roof Edge Detection Process\n\n");
printf("Input filename: ");
gets(IPname);
IPfile=IPname;
printf("Roof Edge Map output filename : ");
gets(OPname);
OPfile=OPname;

do {
   printf("Projection width (odd no.[1-%d]) : 1\b",MaxWidth);
   gets(temp);
   ascii=*temp;
   if (ascii == 0)
       Pjsize=1;
   else
       Pjsize=atoi(temp);
   }
while ((Pjsize <= 0) || (Pjsize > MaxWidth) || (Pjsize/2 == Pjsize/2.0));

do {
   printf("No. of projected data (odd no.[3-%d]) : 7\b",NumPoint);
   gets(temp);
   ascii=*temp;
   if (ascii == 0)
       M=7;
   else
       M=atoi(temp);
   }
while ((M < 3) || (M > NumPoint) || (M/2 == M/2.0));

V=M+2;
do {
   printf("Regularizing factor (>0) : 0.1\b");
   gets(temp);
   ascii=*temp;
   if (ascii == 0)
     B=0.1;
   else
     B=atof(temp);
   }
while (B <= 0);
```

```
do {
   printf("Threshold of the second derivative[0-%d] : 4\b",Tmax);
   gets(temp);
   ascii=*temp;
   if (ascii == 0)
      threshold=4;
   else
      threshold=atoi(temp);
   }
while ((threshold < 0) || (threshold > Tmax));

do {
   printf("Zero: 0\b",Tmax);
   gets(temp);
   ascii=*temp;
   if (ascii == 0)
      Zero=0;
   else
      Zero=atoi(temp);
   }
while (0);

readimage();

}

/* ********************************************************************** */
/* SAVE THE IMAGE TO A FILE AND DISPLAY SOME IMPORTANT INFORMATIONS */
/* ********************************************************************** */
OPproc()
{
writeimage(OPfile,1);

system("clear");
printf("The information for the image and detection\n");
printf("-------------------------------------------\n");
printf("Input  filename : %s\n",IPfile);
printf("Roof Edge Map : %s\n",OPfile);
printf("Pjsize = %d\n",Pjsize);
printf("Medium Regularization Factor , B = %3.1f\n",B);
printf("Data Pts,M = %d\n",M);
printf("Threshold  = %d\n",threshold);
printf("ras_width  = %d\n",ras_width);
printf("ras_height = %d\n",ras_height);
printf("\n");
printf("---*** Finish ! ***--\n");
}

/* ********************************************** */
/* CREATE THE MATRIX OF THE CUBIC B-SPLINE OPERATOR */
/* ********************************************** */
BasicFcts()
{
int   row,col;
char  *temp="\0";
float tmp1[NumPoint],tmp2[NumPoint];

tmp1[0]=0;tmp1[1]=1;tmp1[2]=4;tmp1[3]=1;tmp1[4]=0;
tmp2[0]=0;tmp2[1]=1;tmp2[2]=-2;tmp2[3]=1;tmp2[4]=0;

for (row=0;row<M;row++)
    for (col=0;col<V;col++)
        {
        if (((col-row) > 2) || ((col-row) < 0))
            A0[row][col]=A2[row][col]=0;
```



```
        else
            {
            A0[row][col]=tmp1[1-row+col];
            A2[row][col]=tmp2[1-row+col];
            }
        }
}



/* ********************************************************** */
/* CALCULATE THE ARRAY OF A0 TO THE REQUIRED INTEGERS     */
/* STEP EDGE DETECTION THE ARRAY IS USED WITH A0[0][0]=4  */
/* ********************************************************** */
Calcu_A()
{
int   row,col;

for (row=0;row<M;row++)
    for (col=0;col<V;col++)
        A0[row][col]/=6.0;
 }

/* ********************************************************************** */
/* CALCULATE D=[A]T.A+B[A']T.[A'] WHERE A2 IS THE 2nd DEVIATIVE OF A0 */
/*       [A] IS STORED IN ARRAY A0 AND [A'] IS STORED IN ARRAY A2      */
/* ********************************************************************** */
Calcu_D()
{
int   row,col,cnt;
float temp1[NumPoint+2][NumPoint+2];
float temp2[NumPoint+2][NumPoint+2];

for (row=0;row<V;row++)
    for (col=0;col<V;col++)
        {
        temp1[row][col]=temp2[row][col]=0;
        for (cnt=0;cnt<M;cnt++)
            {
            temp1[row][col]+=(A0[cnt][row]*A0[cnt][col]);
            temp2[row][col]+=(A2[cnt][row]*A2[cnt][col]);
            }
        }
for (row=0;row<V;row++)
    for (col=0;col<V;col++)
        d[row][col]=(temp1[row][col]+B*temp2[row][col]);
}

/* ********************************************************************** */
/* CALCULATE MATRIX=[D]-1.[A]T WHERE THE RESULT IS STORED IN [D] AGAIN */
/* ********************************************************************** */
Calcu_M()
{
int   row,col,num;
float temp[NumPoint+2][NumPoint];

for (row=0;row<V;row++)
        for (col=0;col<M;col++)
        {
        temp[row][col]=0;
        for (num=0;num<V;num++)
            temp[row][col]+=(d[row][num]*A0[col][num]);
        }
for (row=0;row<V;row++)
    for (col=0;col<M;col++)
```

```
        d[row][col]=temp[row][col];
}


/* ******************************************** */
/* CALCULATE THE COEFFICIENT [C]=[D]-1.[A]T.[F] */
/*      STORE THE RESULTS IN ARRAY f AGAIN      */
/* ******************************************** */
GetCoef()
{
int   row,col;
float temp[NumPoint];

for (row=0;row<V;row++)
    {
    temp[row]=0;
    for (col=0;col<M;col++)
        temp[row]+=(d[row][col]*f[col]);
    }
for (row=0;row<V;row++)
    f[row]=temp[row];
}


/* ********************************************************* */
/* READ AN IMAGE FROM A FILE, WHICH MUST BE IN RASTER FORMAT */
/* ********************************************************* */
readimage()
{
int   f1, length, nr, i, j, k, kk, mm;
char  *s;
unsigned char buf[BUFSIZE];

/* open the image file */
f1 = open(IPfile,0);

/* determine various image parameters from raster file header */
for (i=1; i<=8; i++)
    {
    nr = read(f1,buf,4);
    length = buf[0]*256*256*256+buf[1]*256*256+buf[2]*256+buf[3];
    switch (i)
           {
           case 1: ras_magic = length;
                   break;
           case 2: ras_width = length;
                   break;
           case 3: ras_height = length;
                   break;
           case 4: ras_depth = length;
                   break;
           case 5: ras_length = length;
                   break;
           case 6: ras_type = length;
                   break;
           case 7: ras_maptype = length;
                   break;
           case 8: ras_maplength = length;
                   break;
           }
    }

/* skip colormap */
if (ras_maplength > 0)
    lseek(f1,ras_maplength,1);
```



```
/* read image data from file into 2-D array */
i = 0;
kk = 512 / ras_width;
while ((nr = read(f1,buf,512)) > 0)
      {
      mm = 0;
      for (k=1; k <= kk; k++)
          {
          for (j=0; j < ras_width; j++, mm++)
              Timage[i][j] = buf[mm];
          i++;
          }
      }

/* close image file */
close(f1);
}

/* ********************************************************* */
/* WRITE THE PROCESSED IMAGE TO A FILE IN RASTER FORMAT */
/* ********************************************************* */
int iter;
char *OPfilename;
writeimage(OPfilename,iter)
{
int   f2, length, i, j, n1, n2, n3;
unsigned char buf2[BUFSIZE];

/* create a file to store the processed image */
f2 = creat(OPfilename,0644);

/* create header for raster file */
n1 = 256;
n2 = 256 * 256;
n3 = 256 * 256 *256;
for (i=1; i<=8; i++)
    {
    switch (i)
           {
           case 1: length = ras_magic;
                   break;
           case 2: length = ras_width;
                   break;
           case 3: length = ras_height;
                   break;
           case 4: length = ras_depth;
                   break;
           case 5: length = ras_length;
                   break;
           case 6: length = ras_type;
                   break;
           case 7: if (ras_maptype == 2)
                   /* checkboard and crossboard images */
                       length = 1;
                   else
                       length = ras_maptype;
                   break;
           case 8: if (ras_maplength == 0)
                   /* checkboard and crossboard images */
                       length = 768;
                   else
                       length = ras_maplength;
                   break;
           }
    buf2[0] = length / n3;
    buf2[1] = (length - buf2[0]*n3) / n2;
```

```
    buf2[2] = (length - buf2[0]*n3 - buf2[1]*n2) / n1;
    buf2[3] = (length - buf2[0]*n3 - buf2[1]*n2 - buf2[2]*n1);
    write(f2,buf2,4);
    }

/* create the colourmap for monochrome image */
for (i=0; i < 256; i++)
    buf2[i] = i;
write(f2, buf2, 256);   /* red */
write(f2, buf2, 256);   /* green */
write(f2, buf2, 256);   /* blue */

/* write enhanced images into files */
if (iter == 1)
for (i=0; i < ras_height; i++)
    {
    for (j=0; j < ras_width; j++)
        buf2[j] = Map[i][j];
    write(f2,buf2,ras_width);
    }
/*if (iter == 2)
for (i=0; i < ras_height; i++)
    {
    for (j=0; j < ras_width; j++)
        buf2[j] = image2[i][j];
    write(f2,buf2,ras_width);
    }
if (iter == 3)
for (i=0; i < ras_height; i++)
    {
    for (j=0; j < ras_width; j++)
        buf2[j] = image3[i][j];
    write(f2,buf2,ras_width);
    }*/
/* close the output file */
close(f2);
}

/* ************************************************************************** */
/* A MATRIX MUST BE DECOMPOSED INTO ALPHA AND BETA BEFORE FINDING INVERSE */
/*     THIS PROCEDURE IS TO FIND THE INVERSE OF ARRAY DEFINITED AS d      */
/* ************************************************************************** */
Inver_D()
{
LuDcmp();
Inverse();
}

/* ********************************************** */
/* TO DECOMPOSE A MATRIX TO TWO TRIANGULAR MATRIXES */
/*   ONE IS CALLED ALPHA AND ONE IS CALLED BETA     */
/* ********************************************** */
LuDcmp()
{
int   i,j,k;
float temp;

for (j=0;j<V;j++)
    alpha[j][j]=1;
for (j=0;j<V;j++)
    {
    for (i=0;i<=j;i++)
        {
        temp=0;
        for (k=0;k<i;k++)
            temp+=(alpha[i][k]*beta[k][j]);
```



```
        beta[i][j]=d[i][j]-temp;
        }
        if (j < (V-1))
           {
           for (i=j+1;i<V;i++)
               {
               temp=0;
               for (k=0;k<j;k++)
                   temp+=(alpha[i][k]*beta[k][j]);
               if (d[i][j] == temp)
                  alpha[i][j]=0;
               else
                  alpha[i][j]=(d[i][j]-temp)/beta[j][j];
               }
           }
    }
}

/* ************************************************************** */
/* FIND THE INVERSE OF A MATRIX WHICH HAS BEEN DECOMPOSED ALREADY */
/* ************************************************************** */
Inverse()
{
int   row,col,num;
float temp[NumPoint+2][NumPoint+2];

for (col=0;col<V;col++)
    {
    for (num=0;num<V;num++)
        f[num]=0;
    f[col]=1;
    LuFwSb();
    LuBkSb();
    for (num=0;num<V;num++)
        temp[num][col]=f[num];
    }
for (row=0;row<V;row++)
    for (col=0;col<V;col++)
        d[row][col]=temp[row][col];
}

/* ********************** */
/* LU FORWARD SUBSTITUTION */
/* ********************** */
LuFwSb()
{
int   i,j;
float temp;
float y[NumPoint+2];

for (i=0;i<V;i++)
    {
    temp=0;
    for (j=0;j<i;j++)
        temp+=(alpha[i][j]*y[j]);
    y[i]=(f[i]-temp)/alpha[i][i];
    }
for (i=0;i<V;i++)
    f[i]=y[i];
}

/* ********************** */
/* LU BACKWARD SUBSTITUTION */
/* ********************** */
LuBkSb()
{
```

```
int   i,j;
float temp;
float y[NumPoint+2];

for (i=V-1;i>=0;i--)
    {
    temp=0;
    for (j=V-1;j>i;j--)
        temp+=(beta[i][j]*y[j]);
    if (f[i] == temp)
       y[i]=0;
    else
       y[i]=(f[i]-temp)/beta[i][i];
    }
for (i=0;i<V;i++)
    f[i]=y[i];
}


/* ********************************************************** */
/* THIS PROCEDURE IS USED FOR DISPLAYING THE MASK OF A0,A2,d,f */
/* e.g. 1,0,1,0 : Display A0 and d only!!!                    */
/* ********************************************************** */
TestMsk(num1,num2,num3,num4)
{
int   row,col;

/*   if (V == 0)
    max=NumPoint;
else
    max=V;
*/
printf("\n");
if (num1 == 1)
    {
    printf("The array of A0\n");
    printf("~~~~~~~~~~~~~~~\n");
    for (row=0;row<M;row++)
        {
        for (col=0;col<V;col++)
            printf("%8.3f ",A0[row][col]);
        printf("\n");
        }
    }
if (num2 == 1)
    {
    printf("The array of A2\n");
    printf("~~~~~~~~~~~~~~~\n");
    for (row=0;row<M;row++)
        {
        for (col=0;col<V;col++)
            printf("%8.3f ",A2[row][col]);
        printf("\n");
        }
    }
if (num3 == 1)
    {
    printf("The array of d\n");
    printf("~~~~~~~~~~~~~~~\n");
    for (row=0;row<V;row++)
        {
        for (col=0;col<V;col++)
            printf("%8.3f ",d[row][col]);
        printf("\n");
        }
    }
```



```
if (num4 == 1)
    {
    printf("The array of f\n");
    printf("~~~~~~~~~~~~~~~\n");
    for (row=0;row<V;row++)
        printf("%8.3f ",f[row]);
    printf("\n");
    }
}

CUproc()
{

Calcu_D();
Inver_D();
Calcu_M();
}


Initialize()
{
BasicFcts();
TestMsk(1,1);
Calcu_A();
Renewimage();
}

Renewimage()
{
int i,j;


    for(i=0;i<ras_height;i++)
      for(j=0;j<ras_width;j++)
        Map[i][j]=FALSE;

}

GetHoriDeri()
{
int   mid;

mid=V/2;

H1R=0.125*(f[mid+2]+5*f[mid+1]-5*f[mid]-f[mid-1]);
H1L=0.125*(f[mid+1]+5*f[mid]-5*f[mid-1]-f[mid-2]);
H3R=f[mid+2]-3*f[mid+1]+3*f[mid]-f[mid-1];
H3L=f[mid+1]-3*f[mid]+3*f[mid-1]-f[mid-2];
H2=f[mid-1]-2*f[mid]+f[mid+1];
}

GetVerDeri()
{
int   mid;

mid=V/2;
/* Note the sign has to be changed due to the change of coordinate */

V1R=(-0.125)*(f[mid+2]+5*f[mid+1]-5*f[mid]-f[mid-1]);
V1L=(-0.125)*(f[mid+1]+5*f[mid]-5*f[mid-1]-f[mid-2]);
V3R=(-1)*f[mid+2]-3*f[mid+1]+3*f[mid]-f[mid-1];
V3L=(-1)*f[mid+1]-3*f[mid]+3*f[mid-1]-f[mid-2];
V2=f[mid-1]-2*f[mid]+f[mid+1];
}

Fitting()
```

```
{
int i,j,m;

m=M/2;

CUproc();

for(i=m;i<ras_height-m;i++)
      for(j=m;j<ras_width-m;j++)
        {
          GetHori(i,j);
          GetCoef();
          GetHoriDeri();
          GetVer(i,j);
          GetCoef();
          GetVerDeri();

          if ((H3L*H3R<0) || (V3L*V3R<0))
            {
              S2=fabs(H2+V2);
              if (S2>=threshold)
                {
                  if (H1L*H1R<=Zero && V1L*V1R<=Zero)
                    Map[i][j]=TRUE;
                };
            };
        };
};

GetHori(i,j)
{

int r,c,m,n;

m=M/2;

n=Pjsize/2;

for (c=0;c<M;c++)
  {
    f[c]=0;
    for(r=0;r<Pjsize;r++)
      f[c]=f[c]+Timage[i-n+r][j-m+c];
    f[c]=f[c]/Pjsize;
  };
};

GetVer(i,j)
{

int r,c,m,n;

m=M/2;

n=Pjsize/2;

for (r=0;r<M;r++)
  {
    f[r]=0;
    for(c=0;c<Pjsize;c++)
      f[r]=f[r]+Timage[i-m+r][j-n+c];
    f[r]=f[r]/Pjsize;
  };
};
```

```
/* ************************************************************************** */
/* Name of the programme: fcr.c                                               */
/*                                                                            */
/* Function: Calculate the FCR's and ATR's of an edge map                     */
/* ************************************************************************** */
/* To Compile this source code: cc -o fcr fcr.c -lm                           */
/*                                                                            */
/* To run the programme: fcr Ideal_EM Actual_EM DataFile                      */
/*                                                                            */
/* where                                                                      */
/* Ideal_EM  the filename of the ideal edge map                               */
/* Actual_EM the filename of the edge map  produced by an edge detector       */
/* DataFile  the file where the calculated FCR's and ATR's are stored         */
/* ************************************************************************** */


#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define ImagSize   512
#define BUFSIZE    512
#define Edge 0
#define Non_edge 255
#define Tan225 0.41421356
#define Tan675 2.4142136


int   ras_magic, ras_height, ras_type,  ras_maptype;
int   ras_width, ras_length, ras_depth, ras_maplength;
int   ntp, nta, ntm, nne, nfp;
float ratio, ratio2;

char  *IEMfile,*AEMfile, *datafile;
char  ProgName[30];


int   image[ImagSize][ImagSize],image2[ImagSize][ImagSize];

/* ***************************** */
/* MAIN PROGRAM OF EDGE DETECTION */
/* ***************************** */
main(argc,argv)
int  argc;
char *argv[];
{

strcpy (ProgName, argv[0]);
/* handle command line arguments */
if (argc != 4)
{
Usage();
Abort("");
}

IEMfile=argv[1];
AEMfile=argv[2];
datafile=argv[3];
IPproc();

calc_fcr();

OPproc();
}

/* ************************* */
```



```
/* INPUT ALL THE REQUIREMENTS */
/* ************************* */
IPproc()
{

readimage(IEMfile,1);
readimage(AEMfile,2);

}

/* ********************************************************************** */
/* SAVE THE IMAGE TO A FILE AND DISPLAY SOME IMPORTANT INFORMATIONS */
/* ********************************************************************** */
OPproc()
{

FILE *fid;

system("clear");
printf("The information for the False-Correct Ratio\n");
printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
printf("Ideal Edge Map : %s\n",IEMfile);
printf("Actual Edge Map : %s\n",AEMfile);

printf("ras_width  = %d\n",ras_width);
printf("ras_height = %d\n",ras_height);

printf("\n");

printf("True Positive: %d\n",ntp);
printf("True Approximate: %d\n",nta);
printf("True Missing: %d\n",ntm);
printf("Neutral Extra:  %d\n",nne);
printf("False Positive: %d\n",nfp);

printf("\n");

printf("FCR: %.3f\n",ratio);
printf("ATR: %.3f\n",ratio2);
printf("\n");
printf("--*** Finish ! ***--\n");

fid=fopen(datafile,"a");
fprintf(fid,"%s\t",AEMfile);
fprintf(fid,"FCR=%4.2f\t",ratio);
fprintf(fid,"ATR=%4.2f\n\n",ratio2);
fclose(fid);
}

/* ************************************* */

int iter;
char *IPfile;
readimage(IPfile, iter)
{
int   fl, length, nr, i, j, k, kk, mm;
char  *s;
unsigned char buf[BUFSIZE];

/* open the image file */
fl = open(IPfile,0);

/* determine various image parameters from raster file header */
for (i=1; i<=8; i++)
    {
    nr = read(fl,buf,4);
```

```
    length = buf[0]*256*256*256+buf[1]*256*256+buf[2]*256+buf[3];
    switch (i)
           {
           case 1: ras_magic = length;
                   break;
           case 2: ras_width = length;
                   break;
           case 3: ras_height = length;
                   break;
           case 4: ras_depth = length;
                   break;
           case 5: ras_length = length;
                   break;
           case 6: ras_type = length;
                   break;
           case 7: ras_maptype = length;
                   break;
           case 8: ras_maplength = length;
                   break;
           }
    }

/* skip colormap */
if (ras_maplength > 0)
    lseek(f1,ras_maplength,1);

/* read image data from file into 2-D array */

if (iter == 1)
{
i = 0;
kk = 512 / ras_width;
while ((nr = read(f1,buf,512)) > 0)
      {
      mm = 0;
      for (k=1; k <= kk; k++)
          {
          for (j=0; j < ras_width; j++, mm++)
              image[i][j] = buf[mm];
          i++;
          }
      }
};

if (iter == 2)
{
i = 0;
kk = 512 / ras_width;
while ((nr = read(f1,buf,512)) > 0)
      {
      mm = 0;
      for (k=1; k <= kk; k++)
          {
          for (j=0; j < ras_width; j++, mm++)
              image2[i][j] = buf[mm];
          i++;
          }
      }
}
/* close image file */
close(f1);
};

calc_fcr()
{
      int i,j, xc, yc;
```



```
      int x_centre, y_centre;
      int xx, yy;
      float theta;

      ntp=nta=ntm=nne=nfp=0;
      x_centre = ras_width/2;
      y_centre = ras_height/2;



      for (i=0; i < ras_height; i++)
       {
        for (j=0; j < ras_width; j++)
          {
            if (image[i][j]==Edge && image2[i][j]==Edge)
                ntp++;
            else {


                    xc=j-x_centre;
                    yc=y_centre-i;
                    xx=yy=0;
                    if (yc == 0)  yy=1;
                    else {
                      theta=xc/yc;
                      if (theta>(-1)*Tan675 && theta<=(-1)*Tan225)
                        { xx=-1; yy=1;}
                      else {
                        if (theta>(-1)*Tan225 && theta<=Tan225) xx=1;
                        else {
                          if (theta>Tan225 && theta<=Tan675)
                            {xx=1; yy=1;}
                          else yy=1;
                        };
                      };
                     };

                    if (image[i][j]==Edge && image2[i][j]==Non_edge)
                       {
                       if(image2[i+xx][j+yy]==Edge||image2[i-xx][j-yy]==Edge)
                         nta++;
                       else ntm++;
                       };

                    if (image[i][j]==Non_edge && image2[i][j]==Edge)
                       {
                       if(image[i+xx][j+yy]==Edge||image[i-xx][j-yy]==Edge)
                         nne++;
                       else nfp++;
                       };
                  };
          };
        };
        ratio=((ntm+nfp)*1.00)/(nta+ntp);
        ratio2=((nta+nne)*1.00)/ntp;
};

Usage()
{

fprintf (stderr, "Command: %s Ideal_EM Actual_EM DataFile\n", ProgName);

}

Abort(string)
char *string;
```

```
{

if (*string)
fprintf (stderr, "%s ERROR: %s\n", ProgName, string);

exit(3);
}
```

Appendix C

The codes of MRCBS, the Haralick edge detector, Edge Focusing and the Chen/Yang edge detector.

```
/* ************************************************************************ */
/* Name of the programme: meds.c                                            */
/*                                                                          */
/* Function: The multiscale edge detector based on bounded diffusion theory */
/* ************************************************************************ */
/* To Compile this source code: cc -o meds meds.c -lm                       */
/*                                                                          */
/* To run the programme:                                                    */
/* meds InputFile, OutFile, NomFitting, threshold, HF_threshold, Pjsize     */
/*                                                                          */
/* where                                                                    */
/* meds: the name of this executable programme.                             */
/* InputFile: the file name for the input image.                            */
/* OutFile: the file name for the output image in the raster format.        */
/* NomFitting: the number of pixels for the local operation.                */
/* (Recommanded value=13 )                                                  */
/* threshold: the value of the threshold to be used.                        */
/* HF_threshold: the threshold on the high-frequency components.            */
/* (Recommanded value=616)                                                  */
/* Pjsize: the number of pixels for the equal-weighted averaging.           */
/* (Recommanded value=5)                                                    */
/* ************************************************************************ */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* ******************************************** */
/* MaxWidth is the maximum number of Pjsize.    */
/* NumPoint is the maximum number of M.         */
/* ImagSize is the maximum number of image width. */
/* BUFSIZE is the default buffer size.          */
/* WHITEVALUE represents the white grey level.  */
/* BLACKVALUE represents the black grey level.  */
/* ******************************************** */

#define MaxWidth   5
#define NumPoint   15
#define ImagSize   512
#define BUFSIZE    512
#define TRUE       0
#define FALSE      255
#define Tmax       256
#define PI         3.14159263536

int   ras_magic, ras_height, ras_type,  ras_maptype;
int   ras_width, ras_length, ras_depth, ras_maplength;

/* ************************************************** */
/* V : Number of B-splines for curve fitting.         */
/* Pjsize : The width of the projection and the gradient.*/
/* MaskWidth : The width of the submasks for f(i).    */
/* threshold : The value is used for determining edgels. */
/* B : The regularizing factor.                       */
/* ************************************************** */

char ProgName[20];
float   threshold[10];
int   V,M,Pjsize,MaskWidth;
float ratio;
char  *IPfile,*OPfile;
float B;

int   Timage[ImagSize][ImagSize],Map[ImagSize][ImagSize];
float f[NumPoint+2],fh[NumPoint+2],fv[NumPoint+2];
float d[10][NumPoint+2][NumPoint+2];
float A0[NumPoint][NumPoint+2],A2[NumPoint][NumPoint+2];
```



```
float alpha[NumPoint+2][NumPoint+2],beta[NumPoint+2][NumPoint+2];
float mask[NumPoint+MaxWidth+2][NumPoint+MaxWidth+2];
float Ethreshold;
float energy;
float angle;
float CosTheta, SinTheta;

/* ***************************** */
/* MAIN PROGRAM OF EDGE DETECTION */
/* ***************************** */
main(argc,argv)
int  argc;
char *argv[];
{

strcpy (ProgName, argv[0]);
/* handle command line arguments */
if (argc != 7)
{
Usage();
Abort("");
}

IPfile=argv[1];
OPfile=argv[2];
M=atoi(argv[3]);
V=M+2;
B=5;
threshold[0]=atof(argv[4]);
Ethreshold=atof(argv[5]);
ratio=1;
Pjsize=atoi(argv[6]);;
MaskWidth=M+Pjsize;

readimage();

printf("\n\nInitialize");
Initialize();
printf("\n\nFitting");
Fitting();
printf("\n\nOutput");
OPproc();
}


/* ********************************************************************** */
/* SAVE THE IMAGE TO A FILE AND DISPLAY SOME IMPORTANT INFORMATIONS */
/* ********************************************************************** */
OPproc()
{
writeimage(OPfile);

system("clear");
printf("The information for the image and detection\n");
printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
printf("Input  filename : %s\n",IPfile);
printf("Edge Map : %s\n",OPfile);
printf("Pjsize = %d\n",Pjsize);
printf("Regularization Factor , B = %3.1f\n",B);
printf("Energy = %8.1f\n",Ethreshold);
printf("Data Pts,M = %d\n",M);
printf("Threshold[0]  = %5.2f\n",threshold[0]);
printf("Threshold[9]  = %5.2f\n",threshold[9]);
printf("ras_width  = %d\n",ras_width);
printf("ras_height = %d\n",ras_height);
printf("\n");
```

```
printf("--*** Finish ! ***--\n");
}

/* ************************************************ */
/* CREATE THE MATRIX OF THE CUBIC B-SPLINE OPERATOR */
/* ************************************************ */
BasicFcts()
{
int   row,col;
char  *temp="\0";
float tmp1[NumPoint],tmp2[NumPoint];

tmp1[0]=0;tmp1[1]=1;tmp1[2]=4;tmp1[3]=1;tmp1[4]=0;
tmp2[0]=0;tmp2[1]=1;tmp2[2]=-2;tmp2[3]=1;tmp2[4]=0;

for (row=0;row<M;row++)
    for (col=0;col<V;col++)
        {
        if (((col-row) > 2) || ((col-row) < 0))
            A0[row][col]=A2[row][col]=0;
        else
            {
            A0[row][col]=tmp1[1-row+col];
            A2[row][col]=tmp2[1-row+col];
            }
        }
}



/* ******************************************************* */
/* CALCULATE THE ARRAY OF A0 TO THE REQUIRED INTEGERS      */
/* STEP EDGE DETECTION:  THE ARRAY IS USED WITH A0[0][0]=4*/
/* ******************************************************* */
Calcu_A()
{
int   row,col;

for (row=0;row<M;row++)
    for (col=0;col<V;col++)
        A0[row][col]/=6.0;
 }

/* ********************************************************************* */
/* CALCULATE D=[A]T.A+B[A*]T.[A*] WHERE A2 IS THE 2nd DEVIATIVE OF A0 */
/*       [A] IS STORED IN ARRAY A0 AND [A*] IS STORED IN ARRAY A2       */
/* ********************************************************************* */
Calcu_D()
{
int   row,col,cnt,iter;
float BB;
float temp1[NumPoint+2][NumPoint+2];
float temp2[NumPoint+2][NumPoint+2];

for (row=0;row<V;row++)
    for (col=0;col<V;col++)
        {
        temp1[row][col]=temp2[row][col]=0;
        for (cnt=0;cnt<M;cnt++)
            {
            temp1[row][col]+=(A0[cnt][row]*A0[cnt][col]);
            temp2[row][col]+=(A2[cnt][row]*A2[cnt][col]);
            }
        }
 for (iter=0; iter<10;iter++)
```



```
  {
   BB=B-0.5*iter;
   for (row=0;row<V;row++)
     for (col=0;col<V;col++)
       d[iter][row][col]=(temp1[row][col]+BB*temp2[row][col]);
  }
}

/* ********************************************************************** */
/* CALCULATE MATRIX=[D]-1.[A]T WHERE THE RESULT IS STORED IN [D] AGAIN */
/* ********************************************************************** */
Calcu_M()
{
int   row,col,num,iter;
float temp[NumPoint+2][NumPoint];

for (iter=0;iter<10;iter++)
  {
   for (row=0;row<V;row++)
     for (col=0;col<M;col++)
       {
       temp[row][col]=0;
       for (num=0;num<V;num++)
           temp[row][col]+=(d[iter][row][num]*A0[col][num]);
       }
   for (row=0;row<V;row++)
     for (col=0;col<M;col++)
       d[iter][row][col]=temp[row][col];
  }
}


/* ******************************************* */
/* CALCULATE THE COEFFICIENT [C]=[D]-1.[A]T.[F] */
/*      STORE THE RESULTS IN ARRAY f AGAIN      */
/* ******************************************* */
GetHoriCoef(iter)
{
int   row,col;
float temp[NumPoint];

for (row=0;row<V;row++)
    {
    temp[row]=0;
    for (col=0;col<M;col++)
        temp[row]+=(d[iter][row][col]*fh[col]);
    }
for (row=0;row<V;row++)
    fh[row]=temp[row];
}

GetVerCoef(iter)
{
int   row,col;
float temp[NumPoint];

for (row=0;row<V;row++)
    {
    temp[row]=0;
    for (col=0;col<M;col++)
        temp[row]+=(d[iter][row][col]*fv[col]);
    }
for (row=0;row<V;row++)
    fv[row]=temp[row];
}
```

```
/* ********************************************************** */
/* READ AN IMAGE FROM A FILE, WHICH MUST BE IN RASTER FORMAT */
/* ********************************************************** */
readimage()
{
int   f1, length, nr, i, j, k, kk, mm;
char  *s;
unsigned char buf[BUFSIZE];

/* open the image file */
f1 = open(IPfile,0);

/* determine various image parameters from raster file header */
for (i=1; i<=8; i++)
    {
    nr = read(f1,buf,4);
    length = buf[0]*256*256*256+buf[1]*256*256+buf[2]*256+buf[3];
    switch (i)
           {
           case 1: ras_magic = length;
                   break;
           case 2: ras_width = length;
                   break;
           case 3: ras_height = length;
                   break;
           case 4: ras_depth = length;
                   break;
           case 5: ras_length = length;
                   break;
           case 6: ras_type = length;
                   break;
           case 7: ras_maptype = length;
                   break;
           case 8: ras_maplength = length;
                   break;
           }
    }

/* skip colormap */
if (ras_maplength > 0)
    lseek(f1,ras_maplength,1);

/* read image data from file into 2-D array */
i = 0;
kk = 512 / ras_width;
while ((nr = read(f1,buf,512)) > 0)
      {
      mm = 0;
      for (k=1; k <= kk; k++)
          {
          for (j=0; j < ras_width; j++, mm++)
              Timage[i][j] = buf[mm];
          i++;
          }
      }

/* close image file */
close(f1);
}

/* ********************************************************** */
/* WRITE THE PROCESSED IMAGE TO A FILE IN RASTER FORMAT */
/* ********************************************************** */
char *OPfilename;
writeimage(OPfilename)
{
```



```
int   f2, length, i, j, n1, n2, n3;
unsigned char buf2[BUFSIZE];

/* create a file to store the processed image */
f2 = creat(OPfilename,0644);

/* create header for raster file */
n1 = 256;
n2 = 256 * 256;
n3 = 256 * 256 *256;
for (i=1; i<=8; i++)
    {
    switch (i)
           {
           case 1: length = ras_magic;
                   break;
           case 2: length = ras_width;
                   break;
           case 3: length = ras_height;
                   break;
           case 4: length = ras_depth;
                   break;
           case 5: length = ras_length;
                   break;
           case 6: length = ras_type;
                   break;
           case 7: if (ras_maptype == 2)
                   /* checkboard and crossboard images */
                       length = 1;
                   else
                       length = ras_maptype;
                   break;
           case 8: if (ras_maplength == 0)
                   /* checkboard and crossboard images */
                       length = 768;
                   else
                       length = ras_maplength;
                   break;
           }
    buf2[0] = length / n3;
    buf2[1] = (length - buf2[0]*n3) / n2;
    buf2[2] = (length - buf2[0]*n3 - buf2[1]*n2) / n1;
    buf2[3] = (length - buf2[0]*n3 - buf2[1]*n2 - buf2[2]*n1);
    write(f2,buf2,4);
    }

/* create the colourmap for monochrome image */
for (i=0; i < 256; i++)
    buf2[i] = i;
write(f2, buf2, 256);   /* red */
write(f2, buf2, 256);   /* green */
write(f2, buf2, 256);   /* blue */

/* write enhanced images into files */
for (i=0; i < ras_height; i++)
    {
    for (j=0; j < ras_width; j++)
        buf2[j] = Map[i][j];
    write(f2,buf2,ras_width);
    }
close(f2);
}

/* ********************************************************************** */
/* A MATRIX MUST BE DECOMPOSED INTO ALPHA AND BETA BEFORE FINDING INVERSE */
/*     THIS PROCEDURE IS TO FIND THE INVERSE OF ARRAY DEFINITED AS d      */
```

```
/* ************************************************************************ */
Inver_D()
{
int i;
for (i=0;i<10;i++)
  {
    LuDcmp(i);
    Inverse(i);
  }
}

/* ********************************************** */
/* TO DECOMPOSE A MATRIX TO TWO TRIANGULAR MATRIX */
/*   ONE IS CALLED ALPHA AND ONE IS CALLED BETA   */
/* ********************************************** */
LuDcmp(iter)
{
int   i,j,k;
float temp;

for (j=0;j<V;j++)
    alpha[j][j]=1;
for (j=0;j<V;j++)
    {
    for (i=0;i<=j;i++)
        {
        temp=0;
        for (k=0;k<i;k++)
            temp+=(alpha[i][k]*beta[k][j]);
        beta[i][j]=d[iter][i][j]-temp;
        }
        if (j < (V-1))
           {
           for (i=j+1;i<V;i++)
               {
               temp=0;
               for (k=0;k<j;k++)
                   temp+=(alpha[i][k]*beta[k][j]);
               if (d[iter][i][j] == temp)
                  alpha[i][j]=0;
               else
                  alpha[i][j]=(d[iter][i][j]-temp)/beta[j][j];
               }
           }
    }
}

/* ************************************************************* */
/* FIND THE INVERSE OF A MATRIX WHICH HAS BEEN DECOMPOSED ALREADY */
/* ************************************************************* */
Inverse(iter)
{
int   row,col,num;
float temp[NumPoint+2][NumPoint+2];

for (col=0;col<V;col++)
    {
    for (num=0;num<V;num++)
        f[num]=0;
    f[col]=1;
    LuFwSb();
    LuBkSb();
    for (num=0;num<V;num++)
        temp[num][col]=f[num];
    }
for (row=0;row<V;row++)
```



```
    for (col=0;col<V;col++)
        d[iter][row][col]=temp[row][col];
}

/* ********************** */
/* LU FORWARD SUBSTITUTION */
/* ********************** */
LuFwSb()
{
int   i,j;
float temp;
float y[NumPoint+2];

for (i=0;i<V;i++)
    {
    temp=0;
    for (j=0;j<i;j++)
        temp+=(alpha[i][j]*y[j]);
    y[i]=(f[i]-temp)/alpha[i][i];
    }
for (i=0;i<V;i++)
    f[i]=y[i];
}

/* ********************** */
/* LU BACKWARD SUBSTITUTION */
/* ********************** */
LuBkSb()
{
int   i,j;
float temp;
float y[NumPoint+2];

for (i=V-1;i>=0;i--)
    {
    temp=0;
    for (j=V-1;j>i;j--)
        temp+=(beta[i][j]*y[j]);
    if (f[i] == temp)
       y[i]=0;
    else
       y[i]=(f[i]-temp)/beta[i][i];
    }
for (i=0;i<V;i++)
    f[i]=y[i];
}


/* ********************************************************** */
/* THIS PROCEDURE IS USED FOR DISPLAYING THE MASK OF A0,A2,d,f */
/* e.g. 1,0,1,0 : Display A0 and d only!!!                    */
/* ********************************************************** */
TestMsk(num1,num2,num3,num4)
{
int   row,col;

/*   if (V == 0)
    max=NumPoint;
else
    max=V;
*/
printf("\n");
if (num1 == 1)
    {
    printf("The array of A0\n");
    printf("~~~~~~~~~~~~~~~\n");
```

```
    for (row=0;row<M;row++)
        {
        for (col=0;col<V;col++)
            printf("%3.2f ",A0[row][col]);
        printf("\n");
        }
    }
if (num2 == 1)
    {
    printf("The array of A2\n");
    printf("~~~~~~~~~~~~~~~\n");
    for (row=0;row<M;row++)
        {
        for (col=0;col<V;col++)
            printf("%3.2f ",A2[row][col]);
        printf("\n");
        }
    }
if (num3 == 1)
    {
    printf("The array of d\n");
    printf("~~~~~~~~~~~~~~\n");
    for (row=0;row<V;row++)
        {
        for (col=0;col<V;col++)
            printf("%3.2f ",d[row][col]);
        printf("\n");
        }
    }
if (num4 == 1)
    {
    printf("The array of f\n");
    printf("~~~~~~~~~~~~~~\n");
    for (row=0;row<V;row++)
        printf("%3.2f ",f[row]);
    printf("\n");
    }
}

/* ********************************************************** */

CUproc()
{
Calcu_D();
Inver_D();
Calcu_M();
}

/* ********************************************************** */

Initialize()
{
BasicFcts();
TestMsk(1,1);
Calcu_A();
Renewimage();
MakeThreshold();
Ethreshold=Ethreshold*Ethreshold;
}

/* ********************************************************** */

Renewimage()
{
int i,j;
```



```
    for(i=0;i<ras_height;i++)
      for(j=0;j<ras_width;j++)
        Map[i][j]=FALSE;

}


/* ********************************************************** */

Fitting()
{
int i,j,m,nonstop,iter,smallE, mid;
float H1,V1,S1;
float S2,S2L,S2R;

m=MaskWidth/2;
mid=V/2;
CUproc();

for(i=m;i<ras_height-m;i++)
  for(j=m;j<ras_width-m;j++)
    {
        iter=0;
        nonstop=1;
        do
          {
            GetHori(i,j);
            GetHoriCoef(iter);
            H1=0.5*(fh[mid+1]-fh[mid-1]);
            GetVer(i,j);
            GetVerCoef(iter);
            V1=0.5*(fv[mid+1]-fv[mid-1]);
            /* Note it is the i-j coordinate*/

            S1=sqrt(H1*H1+V1*V1);
            if (S1<threshold[iter])
              nonstop=0;

            TestEnergy();

            iter=iter+1;
          }
        while (nonstop && (energy<Ethreshold) && (iter<10));
        if (nonstop)
          {
            iter--;
            CosTheta=H1/S1;
            SinTheta=V1/S1;
            CalData(i,j);
            GetCoef(iter);
            S1=0.5*(f[mid+1]-f[mid-1]);

            if (fabs(S1)>threshold[iter])
              {
                S2=f[mid-1]-2*f[mid]+f[mid+1];
                if (S2==0.0) Map[i][j]=TRUE;
                else
                  {
                    S2R=f[mid]+f[mid+2]-2*f[mid+1];
                    if (S2*S2R<0 && (ratio*fabs(S2R))>=fabs(S2))
                      Map[i][j]=TRUE;
                    else
                      {
                        S2L=f[mid-2]+f[mid]-2*f[mid-1];
                        if (S2*S2L<0 && (ratio*fabs(S2L))>=fabs(S2))
```

```
                        Map[i][j]=TRUE;
                    };
                };
            };

        };
    };
};

/* ********************************************************** */

GetHori(i,j)
{
/* j orientation */
int r,c,m;

m=M/2;

for (c=0;c<M;c++)
  fh[c]=Timage[i][j-m+c];
};

/* ********************************************************** */

GetVer(i,j)
{
/* i orientation */
int r,c,m;

m=M/2;

for (r=0;r<M;r++)
  fv[r]=Timage[i-m+r][j];
};

/* ********************************************************** */

TestEnergy()
{
int   row,col;
float temp[NumPoint];

energy=0;
for (row=0;row<M;row++)
  {
    temp[row]=0;
    for (col=0;col<V;col++)
      temp[row]=temp[row]+A2[row][col]*fh[col];
    energy=energy+temp[row]*temp[row];
  };

for (row=0;row<M;row++)
  {
    temp[row]=0;
    for (col=0;col<V;col++)
      temp[row]=temp[row]+A2[row][col]*fv[col];
    energy=energy+temp[row]*temp[row];
  };

};

/* ********************************************************** */

MakeThreshold()
{
```



```
threshold[1]=threshold[0]*1.0252;
threshold[2]=threshold[0]*1.0537;
threshold[3]=threshold[0]*1.0864;
threshold[4]=threshold[0]*1.1247;
threshold[5]=threshold[0]*1.1706;
threshold[6]=threshold[0]*1.2278;
threshold[7]=threshold[0]*1.3033;
threshold[8]=threshold[0]*1.4136;
threshold[9]=threshold[0]*1.6139;

};

/* ********************************************************** */

Usage()
{

fprintf (stderr, "Command:\n");
fprintf (stderr, "%s InputFile, OutFile, NomFitting, threshold, HF_threshold, Pjsize\
n", ProgName);

}

Abort(string)
char *string;
{

if (*string)
fprintf (stderr, "%s ERROR: %s\n", ProgName, string);

exit(3);
}

/* ************************ */
/* CALCULATE THE DATA POINTS */
/* ************************ */
CalData(r,c)
{
int   CntDat;


for (CntDat=0;CntDat<M;CntDat++)
    {
    GetMask(CntDat);
    Convolu(r,c,CntDat);
    }
}

/* ********************************************************** */
/* OBTAINED THE VALUES OF THE SUBMASKS ACCORDING TO THE ANGLE */
/* ********************************************************** */
GetMask(num)
{
int   row,col;
int   p,m;
int   MaxJ,MaxI;
float TmpJ,TmpI;
float A1,A2,A3,A4;


for (row=0;row<MaskWidth;row++)
    for (col=0;col<MaskWidth;col++)
        mask[row][col]=0;


m=M/2;
```

```
m=num-m;

for (p=-Pjsize/2;p<=Pjsize/2;p++)
    {
    MaxJ=TmpJ=m*CosTheta-p*SinTheta;
    MaxI=TmpI=m*SinTheta+p*CosTheta;
    if (MaxJ < TmpJ)
        MaxJ++;
    if (MaxI < TmpI)
        MaxI++;

    /*  ---------  */
    /* | A1 | A2 | */
    /*  ----+----  */
    /* | A4 | A3 | */
    /*  ---------  */
    A4=fabs((1-fabs(TmpJ-MaxJ))*(TmpI-MaxI));
    A3=fabs((TmpJ-MaxJ)*(TmpI-MaxI));
    A2=fabs((TmpJ-MaxJ)*(1-fabs(TmpI-MaxI)));
    A1=fabs((1-fabs(TmpJ-MaxJ))*(1-fabs(TmpI-MaxI)));
    col=(MaxJ-1)+MaskWidth/2;
    row=MaskWidth/2+MaxI;
    mask[row][col]+=A2;
    col=MaxJ+MaskWidth/2;
    row=MaskWidth/2+MaxI;
    mask[row][col]+=A1;
    col=MaxJ+MaskWidth/2;
    row=MaskWidth/2+(MaxI-1);
    mask[row][col]+=A4;
    col=(MaxJ-1)+MaskWidth/2;
    row=MaskWidth/2+(MaxI-1);
    mask[row][col]+=A3;
    }
}

/* ************************************ */
/* CONVOLUTE THE SUBMASKS WITH THE IMAGE */
/* ************************************ */
Convolu(r,c,num)
{
int   row,col,mid;

mid=MaskWidth/2;
f[num]=0;
for (row=0;row<MaskWidth;row++)
    for (col=0;col<MaskWidth;col++)
        f[num]+=mask[row][col]*Timage[r-mid+row][c-mid+col];
f[num]/=Pjsize;
}


GetCoef(iter)
{
int   row,col;
float temp[NumPoint+2];

for (row=0;row<V;row++)
  {
    temp[row]=0;
    for (col=0;col<M;col++)
        temp[row]+=(d[iter][row][col]*f[col]);
  };

for (row=0;row<V;row++)
  f[row]=temp[row];
}
```

```
/* ************************************************************************** */
/* Name of the programme: facet.c                                             */
/*                                                                            */
/* Function: the Haralick edge detector using the facet model                 */
/* ************************************************************************** */
/* To Compile this source code: cc -o facet facet.c -lm                       */
/*                                                                            */
/* To run the programme: facet rho input-image outfile                        */
/*                                                                            */
/* where                                                                      */
/* rho the allowable subpixel displacement (better smaller than 0.5)          */
/* input_image the filename of the input image                                */
/* outfile the filename of the output image                                   */
/* ************************************************************************** */

#include <stdio.h>
#include <math.h>

#define  degree00  0
#define  degree45  1
#define  degree90  2
#define  degree135 3
#define  degreenil 255
#define  Radian    57.2957795  /* 1 Gadian = 57.2957795 degree */
#define  tan22_5   0.41421356  /* tan(22.5) = 0.41421356 */
#define  tan67_5   2.41421356  /* tan(67.5) = 2.41421356 */
char             ProgName[40],
                 Clut[256][3];   /* color lookup table           */

unsigned char    *Buff,          /* input buffer                 */
                 *OutBuff,       /* output buffer                */
                 *ScanLine2,     /* 2nd scanline                 */
                 *ScanLineN;     /* last scanline                */
unsigned char    *anglebuf,    /* angle of gradient direction  */
                *anglebuf_BAK;

int              Row,            /* # of rows in image           */
                 Col,            /* # of cols in image           */
                 Line;           /* # bytes per scanline         */

float            Rho;            /* subpixel distance            */

 /* directional derivative data */
#define WIDTH 7
#define WIDTH2 49
#define RADIUS 3
#define NUMPOLY 9
#define INFINITY 1.0e5

#define NOT_AN_EDGE 255
#define EDGE 0
int      Polynomial[][WIDTH2] = {
 /* P1 */       {   -3,   -3,   -3,   -3,   -3,   -3,   -3,
                    -2,   -2,   -2,   -2,   -2,   -2,   -2,
                    -1,   -1,   -1,   -1,   -1,   -1,   -1,
                     0,    0,    0,    0,    0,    0,    0,
                     1,    1,    1,    1,    1,    1,    1,
                     2,    2,    2,    2,    2,    2,    2,
                     3,    3,    3,    3,    3,    3,    3},
 /* P2 */       {   -3,   -2,   -1,    0,    1,    2,    3,
                    -3,   -2,   -1,    0,    1,    2,    3,
                    -3,   -2,   -1,    0,    1,    2,    3,
                    -3,   -2,   -1,    0,    1,    2,    3,
                    -3,   -2,   -1,    0,    1,    2,    3,
                    -3,   -2,   -1,    0,    1,    2,    3,
                    -3,   -2,   -1,    0,    1,    2,    3},
```



```
 /* P3 */       {    5,    5,    5,    5,    5,    5,    5,
                     0,    0,    0,    0,    0,    0,    0,
                    -3,   -3,   -3,   -3,   -3,   -3,   -3,
                    -4,   -4,   -4,   -4,   -4,   -4,   -4,
                    -3,   -3,   -3,   -3,   -3,   -3,   -3,
                     0,    0,    0,    0,    0,    0,    0,
                     5,    5,    5,    5,    5,    5,    5},
 /* P4 */       {    9,    6,    3,    0,   -3,   -6,   -9,
                     6,    4,    2,    0,   -2,   -4,   -6,
                     3,    2,    1,    0,   -1,   -2,   -3,
                     0,    0,    0,    0,    0,    0,    0,
                    -3,   -2,   -1,    0,    1,    2,    3,
                    -6,   -4,   -2,    0,    2,    4,    6,
                    -9,   -6,   -3,    0,    3,    6,    9},
 /* P5 */       {    5,    0,   -3,   -4,   -3,    0,    5,
                     5,    0,   -3,   -4,   -3,    0,    5,
                     5,    0,   -3,   -4,   -3,    0,    5,
                     5,    0,   -3,   -4,   -3,    0,    5,
                     5,    0,   -3,   -4,   -3,    0,    5,
                     5,    0,   -3,   -4,   -3,    0,    5,
                     5,    0,   -3,   -4,   -3,    0,    5},
 /* P6 */       {   -1,   -1,   -1,   -1,   -1,   -1,   -1,
                     1,    1,    1,    1,    1,    1,    1,
                     1,    1,    1,    1,    1,    1,    1,
                     0,    0,    0,    0,    0,    0,    0,
                    -1,   -1,   -1,   -1,   -1,   -1,   -1,
                    -1,   -1,   -1,   -1,   -1,   -1,   -1,
                     1,    1,    1,    1,    1,    1,    1},
 /* P7 */       {  -15,  -10,   -5,    0,    5,   10,   15,
                     0,    0,    0,    0,    0,    0,    0,
                     9,    6,    3,    0,   -3,   -6,   -9,
                    12,    8,    4,    0,   -4,   -8,  -12,
                     9,    6,    3,    0,   -3,   -6,   -9,
                     0,    0,    0,    0,    0,    0,    0,
                   -15,  -10,   -5,    0,    5,   10,   15},
 /* P8 */       {  -15,    0,    9,   12,    9,    0,  -15,
                   -10,    0,    6,    8,    6,    0,  -10,
                    -5,    0,    3,    4,    3,    0,   -5,
                     0,    0,    0,    0,    0,    0,    0,
                     5,    0,   -3,   -4,   -3,    0,    5,
                    10,    0,   -6,   -8,   -6,    0,   10,
                    15,    0,   -9,  -12,   -9,    0,   15},
 /* P9 */       {   -1,    1,    1,    0,   -1,   -1,    1,
                    -1,    1,    1,    0,   -1,   -1,    1,
                    -1,    1,    1,    0,   -1,   -1,    1,
                    -1,    1,    1,    0,   -1,   -1,    1,
                    -1,    1,    1,    0,   -1,   -1,    1,
                    -1,    1,    1,    0,   -1,   -1,    1,
                    -1,    1,    1,    0,   -1,   -1,    1}
};

float   Weight[] = {
                196.0,          196.0,          588.0,          784.0,
                588.0,          252.0,         2352.0,         2352.0,
                252.0
};


 /*
  * main - main procedure for program 'dder7'.
  */

main(argc,argv)
int     argc;
char    *argv[];
{
```

```
    int     i, fd1, fd2;

    strcpy (ProgName, argv[0]);

 /* handle cmd line args */
    if (argc != 4) {
        Usage ();
        Abort ("");
    }

    Rho = atof (argv[1]);
    if ((Rho <= 0.0) || (Rho > 1.0))
        Abort ("Rho must be between 0.0 and 1.0");

    if ((fd1 = open(argv[2], 0644)) == -1)
        Abort ("could not open input file");

    if ((fd2 = creat(argv[3], 0644)) == -1)
        Abort ("could not create output file");

 /* read in initial image header */
    rhead(fd1, &Row, &Col);

    if (Row < WIDTH)
        Abort ("# of rows in image must be greater than mask width");

    Line = Col + RADIUS + RADIUS;

 /* alloc WIDTH scanlines for input and 1 scanline for output */
    Buff = (unsigned char *) malloc (Line * WIDTH);
    OutBuff = (unsigned char *) malloc (Col);
    if ((Buff == NULL) || (OutBuff == NULL))
        Abort ("cannot allocate image buffer memory");

    if((anglebuf = (unsigned char *) malloc (Col * Row)) == NULL)
      Abort ("cannot allocate image buffer memory");
    anglebuf_BAK = anglebuf;

    ScanLine2 = Buff + Line;
    ScanLineN = Buff + (WIDTH - 1) * Line;

    whead(fd2, Row, Col);

 /* process images */
    DirDeriv (fd1, fd2);
  /*  output_angle(fd2);*/
    close(fd1);
    close(fd2);

    Abort ("");
}                               /* main */

 /*
  * DirDeriv - process an image, a scanline at a time.
  */

DirDeriv(fd1, fd2)
int fd1, fd2;
{
    int     r;
    unsigned char  *buff;

 /* clear WIDTH scanlines */
    ClrLine (Buff, WIDTH);

 /* read in RADIUS scanlines of data */
```



```
    buff = Buff + (RADIUS + 1) * Line + RADIUS;
    r = RADIUS;
    do {
        FRead (fd1, buff, Col);
        buff += Line;
    } while (--r);

    buff = Buff + (RADIUS + 1) * Line;
    r = RADIUS;
    do {
    ScrollBuff(buff, buff - r* Line,  1);
    } while (--r);

 /* buff -> last scanline */
    buff = ScanLineN + RADIUS;

 /* process image a scanline at a time */
    r = Row;
    do {
    /* scroll data up one scanline */
        ScrollBuff (ScanLine2, Buff, WIDTH - 1);

    /* read in one scanline of data or clear last scanline */
        if (r > RADIUS)
            FRead (fd1, buff, Col);
    /* process scanline */
        DeriveLine ();

    /* write out a scanline */
        FWrite (fd2, OutBuff, Col);
    } while (--r);
}                               /* DirDeriv */

 /*
  * DerivePoint - perform directional derivative operator on
  *               a point;
  */

unsigned char DerivePoint(buff)
unsigned char   *buff;
{
    int     r, c,
            m, sum,
           *p;
    float   k2, k3, k4, k5, k6,
            k7, k8, k9, k10,
            a[NUMPOLY],
           *aptr,
           *w;
    float   sin_a, cos_a,
            sin2_a, cos2_a,
            sin3_a, cos3_a,
            denom,
            x, y, z,
            rho,
            f, fff;
    unsigned char  *bptr;
    float   abstan, tan_a;

 /* calculate 'a' coefficients */
    p = &Polynomial[0][0];
    w = Weight;
    aptr = a;
    m = NUMPOLY;
    do {
        bptr = buff;
```

```
        sum = 0;
        r = WIDTH;
        do {
            c = WIDTH;
            do
                sum += *(bptr++) * *(p++);
            while (--c);
            bptr += (Col - 1);
        } while (--r);
        *(aptr++) = ((float) sum) / *(w++);
    } while (--m);

/* calculate K coefficients */
    k2 = a[0] - (196.0 / 28.0) * a[5] - (28.0 / 7.0) * a[7];
    k3 = a[1] - (28.0 / 7.0) * a[6] - (196.0 / 28.0) * a[8];
    k4 = a[2];
    k5 = a[3];
    k6 = a[4];
    k7 = a[5];
    k8 = a[6];
    k9 = a[7];
    k10 = a[8];

/* calculate sin and cos values */
    denom = sqrt (k2 * k2 + k3 * k3);
    if (denom == 0.0) {
        sin_a = INFINITY;
        cos_a = INFINITY;
    }
    else {
        sin_a = k2 / denom;
        cos_a = k3 / denom;
    }
    sin2_a = sin_a * sin_a;
    cos2_a = cos_a * cos_a;
    sin3_a = sin_a * sin2_a;
    cos3_a = cos_a * cos2_a;

 /*
  * calculate derivatives.
  * f'   = x*rho*rho + y*rho + z
  * f''  = 2*x*rho + y
  * f''' = 2*x
  */

    x = 3.0 * (k7 * sin3_a + k8 * sin2_a * cos_a +
            k9 * sin_a * cos2_a + k10 * cos3_a);
    y = 2.0 * (k4 * sin2_a + k5 * sin_a * cos_a + k6 * cos2_a);
    z = k2 * sin_a + k3 * cos_a;

    *anglebuf  = degreenil;

    if (x == 0.0)
        return (NOT_AN_EDGE);
    else { rho = -y / (2.0 * x);
           if (fabs (rho) >= Rho)
               return (NOT_AN_EDGE);
           else{ fff = 2.0 * x;
                 if (fff >= 0.0)
                   return (NOT_AN_EDGE);
                 else return (EDGE);
               };
         };

}                               /* DerivePoint */
```



```
 /*
  * DeriveLine - perform directional derivative operater on a
  *              scanline of data.
  */

DeriveLine()
{
    int     c;
    unsigned char  *buff,
                   *outbuff;

    buff = Buff;
    outbuff = OutBuff;
    c = Col;
    do{
        *(outbuff++) = DerivePoint (buff++);
        anglebuf++;
       }
    while (--c);
}                               /* DeriveLine */

 /*
  * ClrLine - clear 'num' scanlines starting at 'ptr'.
  */

ClrLine(ptr,num)
unsigned char   *ptr;
int             num;
{
    register int    numbytes;

    numbytes = num * Line;
    do
        *(ptr++) = 0;
    while (--numbytes);
}                               /* ClrLine */

 /*
  * ScrollBuff - scroll 'Buff' by several  scanlines.
  */

ScrollBuff(srcbuf, destbuf, numoflines)
unsigned char *srcbuf, *destbuf;
int numoflines;
{
    register int    numbytes;
    register unsigned char *src,
                           *dest;

    src = srcbuf;
    dest = destbuf;
    numbytes = numoflines * Line;
    do
        *(dest++) = *(src++);
    while (--numbytes);
}                               /* ScrollBuff */

 /*
  * Usage - output program Usage.
  */

Usage()
{
    fprintf (stderr, "%s rho input-image outfile\n", ProgName);
    fprintf (stderr, "    rho -> sub-pixel distance acceptance value\n");
```

```
    fprintf (stderr, "    input-image -> input image file\n");
    fprintf (stderr, "    outfile -> output edge map and direction map\n");
}                               /* Usage */

 /*
  * Abort - terminate program with extreme prejudice.
  */

Abort(string)
char    *string;
{
    if (*string)
        fprintf (stderr, "%s ERROR: %s\n", ProgName, string);

    exit (3);
}                               /* Abort */
whead(fd,rows,cols)
int fd,rows,cols;
{
  outint(fd,1504078485);
  outint(fd,cols);
  outint(fd,rows);
  outint(fd,8);
  outint(fd,cols*rows);
  outint(fd,1);
  outint(fd,2);
  outint(fd,0);
}

rhead(fd, pRows, pCols)
int fd, *pRows, *pCols;
{
        unsigned char buf[32];

        if (read(fd, buf, 32)!=32)
                abort("Read Image header error.");

         *pCols=buf[4]*256*256*256+buf[5]*256*256+buf[6]*256+buf[7];
         *pRows=buf[8]*256*256*256+buf[9]*256*256+buf[10]*256+buf[11];
}

outint(fd,m)
int fd,m;
{ unsigned char buf[4];
  int n1,n2,n3,i;
  n1=256;
  n2=256*256;
  n3=256*256*256;
  buf[0]=m/n3;
  buf[1]=(m-buf[0]*n3)/n2;
  buf[2]=(m-buf[0]*n3-buf[1]*n2)/n1;
  buf[3]=m-buf[0]*n3-buf[1]*n2-buf[2]*n1;
  if (write(fd,buf,4)!=4) abort("writing error\n");
}

abort(string)
char    *string;
{
        if(*string)
                fprintf(stderr, "%s \n", string);
        exit(1);
}
 /*
  * FRead - read 'numbytes' to 'pointer' and check for errors.
  */
```



```
FRead(fd, pointer,numbytes)
char    *pointer;
int     fd, numbytes;
{
    char *pbuf1, *pbuf2;
    int i;

    if (read (fd, pointer, numbytes) != numbytes)
        Abort ("reached end of input file");

/* pad the image */

    pbuf1 = pointer - RADIUS;
    pbuf2 = pointer + Col;

    for (i=0; i<RADIUS; i++)
    {
    *pbuf1++ = pointer[0];
    *pbuf2++ = pointer[Col-1];
    }
}                               /* FRead */


 /*
  * FWrite - write 'len' bytes from 'pointer' to stdout, checking for errors.
  */

FWrite(fd, pointer,len)
char    *pointer;
int     len, fd;
{
    if (write (fd, pointer, len) != len)
        Abort ("cannot write to output");
}                               /* FWrite */


output_angle(fd)
{
    int   i;

        for(i = 0; i < Row; i++)
              FWrite(fd, anglebuf_BAK+i*Col, Col);
                  anglebuf = anglebuf_BAK;
}
```

```
/* ************************************************************************ */
/* Name of the programme: ef.c                                              */
/*                                                                          */
/* Function: the Edge Focusing multiscale edge detector                     */
/* ************************************************************************ */
/* To Compile this source code: cc -o ef ef.c -lm                           */
/*                                                                          */
/* To run the programme: ef InputFile, OutFile, threshold, numlevel         */
/*                                                                          */
/* where                                                                    */
/* InputFile: the file name for the input image.                            */
/* OutFile: the file name for the output image in the raster format.        */
/* threshold: the value of the threshold to be used.                        */
/* numlevel: the number of levels (scales).                                 */
/* ************************************************************************ */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* ******************************************** */
/* NumScale is the number of the scale          */
/* ImagSize is the maximum number of image width. */
/* BUFSIZE is the default buffer size.          */
/* ******************************************** */

#define ImagSize   512
#define BUFSIZE    512
#define TRUE       0
#define FALSE      255
#define Tmax       256
#define PI         3.14159263536
#define NumScale   11
#define MaskWidth  40
#define TWOPI      6.283185307
#define ROOT2      1.314213564
#define FOURROOT2  5.256854256 /* 4*sqrt(2) */


int   ras_magic, ras_height, ras_type,  ras_maptype;
int   ras_width, ras_length, ras_depth, ras_maplength;

/* ************************************************** */
/* MaskWidth : The width of the Gaussian Mask.        */
/* threshold : The value is used for determining edgels. */
/* Std[numScale-1] : scale.                           */
/* ************************************************** */

char ProgName[20];
float   threshold;
int   Width, Radius;
char  *IPfile,*OPfile;
int   Timage[ImagSize][ImagSize];
int   Boolean[2][ImagSize][ImagSize];
float Fimage[ImagSize][ImagSize];
float mask1[MaskWidth], mask2[MaskWidth][MaskWidth];
float Std[NumScale]={4.2, 3.85, 3.5, 3.2, 2.8, 2.5, 2.1, 1.75, 1.4, 1.0, 0.7};
float Sd2[NumScale],TwoSd2[NumScale];
int   level, numlevel;

/* ***************************** */
/* MAIN PROGRAM OF EDGE DETECTION */
/* ***************************** */
main(argc,argv)
int  argc;
char *argv[];
```



```
{

strcpy (ProgName, argv[0]);
/* handle command line arguments */
if (argc != 5)
{
Usage();
Abort("");
}

IPfile=argv[1];
OPfile=argv[2];
threshold=atof(argv[3]);
numlevel=atoi(argv[4]);

readimage();
level=0;

printf("\n\n threshold=%f",threshold);

printf("\n\n Coarsest");
Coarsest();
LoG(0);
ZeroCrossing();

printf("\n\n Recursive");

for (level=1;level<numlevel;level++)
{
  printf("\n\n level=%d",level);
  Neighbour();
  printf("\n\n LoG");
  LoG(1);
  printf("\n\n Std=%f",Std[level]);
  printf("\n\n Zero-Crossing");
  ZeroCrossing();
};

printf("\n\nOutput");
writeimage(OPfile);
}

/* MakeMask ************************************* */

MakeMask()
{
    int     index;

    double  index2;

 /* calc Width of mask */
    Width = ceil (FOURROOT2 * Std[level]);
 /* make Width odd */
    if (Width%2==0) Width++;
    Radius = (Width / 2);

    Sd2[level] = Std[level] * Std[level];
    TwoSd2[level] = 2.0 * Sd2[level];

/* because only zero-crossing is concerned, the scaling of the 1-D
Gaussian kernel is omitted */

    for (index = -Radius; index <= Radius; index++) {
        index2 = index * index;
        mask1[index+Radius] = exp (-index2 / TwoSd2[level]);
   /*   printf("\n\n %f",mask1[index+Radius]); */
```

```
    }
}                                   /* MakeMask */

/* Coarsest  ************************************** */

Coarsest()
{

int i,j, index;
float I1, J1, S1;

    MakeMask();

    BooleanReset(0);

    threshold=threshold*ROOT2*Std[0]*Std[0]*Std[0];

    printf("\n\n threshold=%f",threshold);

    for(i=Radius;i<ras_height-Radius;i++)
      for(j=Radius;j<ras_width-Radius;j++)
        {
          I1=J1=S1=0;
          for (index=-Radius;index<=Radius;index++)
            {
              I1=I1-index*mask1[index+Radius]*Timage[i+index][j];
              J1=J1-index*mask1[index+Radius]*Timage[i][j+index];
            };
          S1=sqrt(I1*I1+J1*J1);
          if (S1>=threshold)
            Boolean[0][i][j]=TRUE;
        }
}


/* ********************************************************* */
/* READ AN IMAGE FROM A FILE, WHICH MUST BE IN RASTER FORMAT */
/* ********************************************************* */
readimage()
{
int   f1, length, nr, i, j, k, kk, mm;
char  *s;
unsigned char buf[BUFSIZE];

/* open the image file */
f1 = open(IPfile,0);

/* determine various image parameters from raster file header */
for (i=1; i<=8; i++)
    {
    nr = read(f1,buf,4);
    length = buf[0]*256*256*256+buf[1]*256*256+buf[2]*256+buf[3];
    switch (i)
           {
           case 1: ras_magic = length;
                   break;
           case 2: ras_width = length;
                   break;
           case 3: ras_height = length;
                   break;
           case 4: ras_depth = length;
                   break;
           case 5: ras_length = length;
                   break;
           case 6: ras_type = length;
                   break;
```



```
           case 7: ras_maptype = length;
                   break;
           case 8: ras_maplength = length;
                   break;
           }
    }

/* skip colormap */
if (ras_maplength > 0)
    lseek(f1,ras_maplength,1);

/* read image data from file into 2-D array */
i = 0;
kk = 512 / ras_width;
while ((nr = read(f1,buf,512)) > 0)
      {
      mm = 0;
      for (k=1; k <= kk; k++)
          {
          for (j=0; j < ras_width; j++, mm++)
              Timage[i][j] = buf[mm];
          i++;
          }
      }

/* close image file */
close(f1);
}

/* ********************************************************* */
/* WRITE THE PROCESSED IMAGE TO A FILE IN RASTER FORMAT */
/* ********************************************************* */
char *OPfilename;
writeimage(OPfilename)
{
int   f2, length, i, j, n1, n2, n3;
unsigned char buf2[BUFSIZE];

/* create a file to store the processed image */
f2 = creat(OPfilename,0644);

/* create header for raster file */
n1 = 256;
n2 = 256 * 256;
n3 = 256 * 256 *256;
for (i=1; i<=8; i++)
    {
    switch (i)
           {
           case 1: length = ras_magic;
                   break;
           case 2: length = ras_width;
                   break;
           case 3: length = ras_height;
                   break;
           case 4: length = ras_depth;
                   break;
           case 5: length = ras_length;
                   break;
           case 6: length = ras_type;
                   break;
           case 7: if (ras_maptype == 2)
                   /* checkboard and crossboard images */
                      length = 1;
                   else
                      length = ras_maptype;
```

```
                break;
            case 8: if (ras_maplength == 0)
                /* checkboard and crossboard images */
                    length = 768;
                else
                    length = ras_maplength;
                break;
            }
    buf2[0] = length / n3;
    buf2[1] = (length - buf2[0]*n3) / n2;
    buf2[2] = (length - buf2[0]*n3 - buf2[1]*n2) / n1;
    buf2[3] = (length - buf2[0]*n3 - buf2[1]*n2 - buf2[2]*n1);
    write(f2,buf2,4);
    }

/* create the colourmap for monochrome image */
for (i=0; i < 256; i++)
    buf2[i] = i;
write(f2, buf2, 256);   /* red */
write(f2, buf2, 256);   /* green */
write(f2, buf2, 256);   /* blue */

/* write enhanced images into files */
for (i=0; i < ras_height; i++)
    {
    for (j=0; j < ras_width; j++)
        buf2[j] = Boolean[0][i][j];
    write(f2,buf2,ras_width);
    }
close(f2);
}

/* ********************************************************** */

TestMask()
{
int  i,j;

    printf("\n The Gaussian Mask\n");
    printf("~~~~~~~~~~~~~~~\n");
    for (i=0;i<Width;i++)
        {
        for (j=0;j<Width;j++)
            printf("%6.3f ",mask2[i][j]);
        printf("\n");
        }
}


/* ********************************************************** */
FloatOutput()
{
int i,j;

    for(i=120;i<124;i++)
      {
        printf("%d\n\n",i);
        for(j=70;j<120;j++)
          printf("%7.4f",Fimage[i][j]);
        printf("\n");
      };
}

FloatReset()
{
int i,j;
```



```
    for(i=0;i<ras_height;i++)
      for(j=0;j<ras_width;j++)
        Fimage[i][j]=0;

}

BooleanReset(lnum)
int lnum;
{
int i,j;


    for(i=0;i<ras_height;i++)
      for(j=0;j<ras_width;j++)
        Boolean[lnum][i][j]=FALSE;

}

/* ********************************************************** */


Usage()
{

fprintf (stderr, "Command:\n");
fprintf (stderr, "%s InputFile, OutFile, threshold, numlevel\n", ProgName);

}

Abort(string)
char *string;
{

if (*string)
fprintf (stderr, "%s ERROR: %s\n", ProgName, string);

exit(3);
}

/* Neighbour  ************************************* */

Neighbour()
{

    int i,j;

    BooleanReset(1);

    for(i=Radius;i<ras_height-Radius;i++)
      for(j=Radius;j<ras_width-Radius;j++)
        {
          if (Boolean[0][i][j]==TRUE)
            {

                  Boolean[1][i-1][j-1]=TRUE;
                  Boolean[1][i-1][j]=TRUE;
                  Boolean[1][i-1][j+1]=TRUE;
                  Boolean[1][i][j-1]=TRUE;
                  Boolean[1][i][j]=TRUE;
                  Boolean[1][i][j+1]=TRUE;
                  Boolean[1][i+1][j-1]=TRUE;
                  Boolean[1][i+1][j]=TRUE;
                  Boolean[1][i+1][j+1]=TRUE;
            };
```

```
        };

}

/* LoG  ************************************** */

LoG(gnum)
int gnum;
{

    int i,j,indexi,indexj;

    MakeMask();

    for (i=0;i<Width;i++)
      for (j=0;j<Width;j++)
        mask2[i][j]=mask1[i]*mask1[j];

/*    TestMask(); */

    FloatReset();

    for(i=Radius;i<ras_height-Radius;i++)
      for(j=Radius;j<ras_width-Radius;j++)
        {
          if (Boolean[gnum][i][j]==TRUE)
            for (indexi=-Radius;indexi<=Radius;indexi++)
              for (indexj=-Radius;indexj<=Radius;indexj++)
                Fimage[i][j]=Fimage[i][j]
                  +(1.0*(indexi*indexi+indexj*indexj)-TwoSd2[level])
                    *mask2[indexi+Radius][indexj+Radius]
                      *Timage[i+indexi][j+indexj];

        };

/*  FloatOutput(); */
};

ZeroCrossing()
{

    int i,j;

    BooleanReset(0);

    for(i=Radius;i<ras_height-Radius;i++)
      for(j=Radius;j<ras_width-Radius;j++)
        {
          if (Fimage[i][j]*Fimage[i-1][j]<0
              && fabs(Fimage[i-1][j])>=fabs(Fimage[i][j]))
            Boolean[0][i][j]=TRUE;
          else
            {
              if (Fimage[i][j]*Fimage[i+1][j]<0
                  && fabs(Fimage[i+1][j])>=fabs(Fimage[i][j]))
                Boolean[0][i][j]=TRUE;
              else
                {
                  if (Fimage[i][j]*Fimage[i][j-1]<0
                      && fabs(Fimage[i][j-1])>=fabs(Fimage[i][j]))
                    Boolean[0][i][j]=TRUE;
                  else
                    {
                      if (Fimage[i][j]*Fimage[i][j+1]<0
                          && fabs(Fimage[i][j+1])>=fabs(Fimage[i][j]))
                        Boolean[0][i][j]=TRUE;
```



```
                    };
                };
            };
        };
}
```

```
/* ************************************************************************ */
/* Name of the programme: rcbs.c                                            */
/*                                                                          */
/* Function: the Regularised Cubic B-Spline edge detector                   */
/* ************************************************************************ */
/* To Compile this source code: cc -o rcbs rcbs.c -lm                       */
/*                                                                          */
/* To run the programme:                                                    */
/*     rcbs InputFile, OutFile, scale, Pjsize, NomFitting, threshold.       */
/*                                                                          */
/* where                                                                    */
/* InputFile: the file name for the input image.                            */
/* OutFile: the file name for the output image in the raster format.        */
/* scale: the regularizing factor of alpha (Recommanded value=0.1).         */
/* Pjsize: the number of pixels for the equal-weighted averaging.           */
/* (Recommanded value=3)                                                    */
/* NomFitting: the number of pixels for the local operation.                */
/* (Recommanded value=5)                                                    */
/* threshold: the value of the threshold to be used.                        */
/* ************************************************************************ */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* ********************************************* */
/* rcbs r25 rcbs.r25 1.8 5 7 3                   */
/* MaxWidth is the maximum number of Pjsize.     */
/* NumPoint is the maximum number of M.          */
/* ImagSize is the maximum number of image width. */
/* BUFSIZE is the default buffer size.           */
/* ********************************************* */

#define MaxWidth   5
#define NumPoint   10
#define ImagSize   512
#define BUFSIZE    512
#define TRUE       0
#define FALSE      255
#define Tmax       256

#define PI         3.14159263536

int   ras_magic, ras_height, ras_type,  ras_maptype;
int   ras_width, ras_length, ras_depth, ras_maplength;

/* ************************************************** */
/* V : Number of B-splines for curve fitting.         */
/* Pjsize : The width of the projector and the gradient. */
/* MaskWidth : The width of the submasks for f(i).    */
/* threshold : The value is used for determining edgels. */
/* B : The regularizing factor.                       */
/* Aimag : The angle of Gradient.                     */
/* Mimag : The magitude of Gradient.                  */
/* ************************************************** */

char ProgName[20];
int   V,M,Pjsize,MaskWidth,threshold,threshold2;
char  *IPfile,*OPfile;
char  IPname[15],OPname[15],OPname2[15],OPname3[15];
float B,Aimag,Mimag;
int   Gh[MaxWidth][MaxWidth],Gv[MaxWidth][MaxWidth];
int   Timage[ImagSize][ImagSize];
int   Map[ImagSize][ImagSize];
float f[NumPoint+2],d[NumPoint+2][NumPoint+2];
float A0[NumPoint][NumPoint+2],A2[NumPoint][NumPoint+2];
float alpha[NumPoint+2][NumPoint+2],beta[NumPoint+2][NumPoint+2];
```



```
float mask[NumPoint+MaxWidth+2][NumPoint+MaxWidth+2];
float CosTheta, SinTheta;

/* ****************************** */
/* MAIN PROGRAM OF EDGE DETECTION */
/* ****************************** */
main(argc,argv)
int  argc;
char *argv[];
{

strcpy (ProgName, argv[0]);
/* handle command line arguments */
if (argc != 7)
{
Usage();
Abort("");
}

IPfile=argv[1];
OPfile=argv[2];
B=atof(argv[3]);
Pjsize=atoi(argv[4]);
M=atoi(argv[5]);
V=M+2;
threshold=atof(argv[6]);
threshold2=0.8*(threshold);
printf("\n\n ReadImage");
readimage();
printf("\n\n Initialize");
Initialize();
printf("\n\n Testing");
Test();
printf("\n\n Output");
OPproc();
}


/* ********************************************************************** */
/* SAVE THE IMAGE TO A FILE AND DISPLAY SOME IMPORTANT INFORMATIONS       */
/* ********************************************************************** */
OPproc()
{
writeimage(OPfile);

system("clear");
printf("The information for the image and detection\n");
printf("-----------------------------------------\n");
printf("Input  filename : %s\n",IPfile);
printf("Step Edge Map : %s\n",OPfile);

printf("Pjsize = %d\n",Pjsize);
printf("Regularization Factor , B = %3.1f\n",B);
printf("Data Pts,M = %d\n",M);
printf("Threshold  = %d\n",threshold);
printf("ras_width  = %d\n",ras_width);
printf("ras_height = %d\n",ras_height);

printf("\n");
printf("---*** Finish ! ***---\n");
}

/* ********************************************** */
/* CREATE THE MATRIX OF THE CUBIC B-SPLINE OPERATOR */
/* ********************************************** */
BasicFcts()
```

```
{
int   row,col;
char  *temp="\0";
float tmp1[NumPoint],tmp2[NumPoint];

tmp1[0]=0;tmp1[1]=1;tmp1[2]=4;tmp1[3]=1;tmp1[4]=0;
tmp2[0]=0;tmp2[1]=1;tmp2[2]=-2;tmp2[3]=1;tmp2[4]=0;

for (row=0;row<M;row++)
    for (col=0;col<V;col++)
        {
        if (((col-row) > 2) || ((col-row) < 0))
            A0[row][col]=A2[row][col]=0;
        else
            {
            A0[row][col]=tmp1[1-row+col];
            A2[row][col]=tmp2[1-row+col];
            }
        }
}



/* ********************************* */
/* GENERATE THE MASK OF THE GRADIENT */
/* ********************************* */
GenMask()
{
int    col,row,mid;

mid=Pjsize/2;
for (col=0;col<Pjsize;col++)
    for (row=0;row<Pjsize;row++)
        {
        Gh[row][col]=col-mid;
        Gv[row][col]=mid-row;
        }
}

/* ******************************** */
/* CALCULATE THE VALUE OF GRADIENT */
/* ******************************** */
int CalGrad(r,c)
{
int    col,row,mid;
float  GradVert,GradHori,SS1;

mid=Pjsize/2;
GradVert=0;
GradHori=0;
for (col=0;col<Pjsize;col++)
    for (row=0;row<Pjsize;row++)
        {
        GradVert+=(Gv[row][col]*Timage[r-mid+row][c-mid+col]);
        GradHori+=(Gh[row][col]*Timage[r-mid+row][c-mid+col]);
        }
SS1=sqrt(GradVert*GradVert+GradHori*GradHori);
if (SS1<threshold2)
  return (0);
else
  {
    CosTheta=GradHori/SS1;
    SinTheta=GradVert/SS1;
    return (1);
  };
};
```



```
/* ************************ */
/* CALCULATE THE DATA POINTS */
/* ************************ */
CalData(r,c)
{
int   CntDat;


for (CntDat=0;CntDat<M;CntDat++)
    {
    GetMask(CntDat);
    Convolu(r,c,CntDat);
    }
}

/* ********************************************************************** */
/*                    DETERMINE THE SIZE OF THE SUBMASK                   */
/* THE SIZE OF THE SUBMASK WILL BE DIFFERENT ACCORDING TO M/V AND Pjsize */
/* ********************************************************************** */
SetRange()
{
int   r;
float radius;

r=radius=sqrt((M/2.0)*(M/2.0)+(Pjsize/2.0)*(Pjsize/2.0));
if (radius > r)
   r++;
MaskWidth=r*2+1;
}

/* ********************************************************** */
/* OBTAINED THE VALUES OF THE SUBMASKS ACCORDING TO THE ANGLE */
/* ********************************************************** */
GetMask(m)
{

/* Note RCBS use x-y coordinate */

int   row,col,p;
int   MaxX,MaxY;
float TmpX,TmpY;
float A1,A2,A3,A4;

for (row=0;row<MaskWidth;row++)
    for (col=0;col<MaskWidth;col++)
        mask[row][col]=0;

m-=(M/2);

for (p=-Pjsize/2;p<=Pjsize/2;p++)
    {
    MaxX=TmpX=m*CosTheta-p*SinTheta;
    MaxY=TmpY=m*SinTheta+p*CosTheta;
    if (MaxX < TmpX)
        MaxX++;
    if (MaxY < TmpY)
        MaxY++;

    /*  ---------  */
    /* | A1 | A2 | */
    /*  ----+----  */
    /* | A4 | A3 | */
    /*  ---------  */
    A1=fabs((1-fabs(TmpX-MaxX))*(TmpY-MaxY));
```

```
    A2=fabs((TmpX-MaxX)*(TmpY-MaxY));
    A3=fabs((TmpX-MaxX)*(1-fabs(TmpY-MaxY)));
    A4=fabs((1-fabs(TmpX-MaxX))*(1-fabs(TmpY-MaxY)));
    col=(MaxX-1)+MaskWidth/2;
    row=MaskWidth/2-MaxY;
    mask[row][col]+=A3;
    col=MaxX+MaskWidth/2;
    row=MaskWidth/2-MaxY;
    mask[row][col]+=A4;
    col=MaxX+MaskWidth/2;
    row=MaskWidth/2-(MaxY-1);
    mask[row][col]+=A1;
    col=(MaxX-1)+MaskWidth/2;
    row=MaskWidth/2-(MaxY-1);
    mask[row][col]+=A2;
    }
m+=(M/2);
Aimag*=(180.0/PI);
}

/* ********************************** */
/* CONVOLUTE THE SUBMASKS WITH THE IMAGE */
/* ********************************** */
Convolu(r,c,num)
{
int   row,col,mid;

mid=MaskWidth/2;
f[num]=0;
for (row=0;row<MaskWidth;row++)
    for (col=0;col<MaskWidth;col++)
        f[num]+=mask[row][col]*Timage[r-mid+row][c-mid+col];
f[num]/=Pjsize;
}

/* ****************************************************** */
/* CALCULATE THE ARRAY OF A0 TO THE REQUIRED INTEGERS     */
/* STEP EDGE DETECTION THE ARRAY IS USED WITH A0[0][0]=4  */
/* ****************************************************** */
Calcu_A()
{
int   row,col;

for (row=0;row<M;row++)
    for (col=0;col<V;col++)
        A0[row][col]/=6.0;
 }

/* ********************************************************************** */
/* CALCULATE D=[A]T.A+B[A*]T.[A*] WHERE A2 IS THE 2nd DEVIATIVE OF A0 */
/*       [A] IS STORED IN ARRAY A0 AND [A*] IS STORED IN ARRAY A2      */
/* ********************************************************************** */
Calcu_D()
{
int   row,col,cnt;
float temp1[NumPoint+2][NumPoint+2];
float temp2[NumPoint+2][NumPoint+2];

for (row=0;row<V;row++)
    for (col=0;col<V;col++)
        {
        temp1[row][col]=temp2[row][col]=0;
        for (cnt=0;cnt<M;cnt++)
            {
            temp1[row][col]+=(A0[cnt][row]*A0[cnt][col]);
            temp2[row][col]+=(A2[cnt][row]*A2[cnt][col]);
```



```
            }
        }
for (row=0;row<V;row++)
    for (col=0;col<V;col++)
        d[row][col]=(temp1[row][col]+B*temp2[row][col]);
}

/* ********************************************************************** */
/* CALCULATE MATRIX=[D]-1.[A]T WHERE THE RESULT IS STORED IN [D] AGAIN */
/* ********************************************************************** */
Calcu_M()
{
int   row,col,num;
float temp[NumPoint+2][NumPoint];

for (row=0;row<V;row++)
        for (col=0;col<M;col++)
        {
        temp[row][col]=0;
        for (num=0;num<V;num++)
            temp[row][col]+=(d[row][num]*A0[col][num]);
        }
for (row=0;row<V;row++)
    for (col=0;col<M;col++)
        d[row][col]=temp[row][col];
}


/* ****************************************** */
/* CALCULATE THE COEFFICIENT [C]=[D]-1.[A]T.[F] */
/*       STORE THE RESULTS IN ARRAY f AGAIN      */
/* ****************************************** */
GetCoef()
{
int   row,col;
float temp[NumPoint];

for (row=0;row<V;row++)
    {
    temp[row]=0;
    for (col=0;col<M;col++)
        temp[row]+=(d[row][col]*f[col]);
    }
for (row=0;row<V;row++)
    f[row]=temp[row];
}


/* ********************************************************** */
/* READ AN IMAGE FROM A FILE, WHICH MUST BE IN RASTER FORMAT */
/* ********************************************************** */
readimage()
{
int   fl, length, nr, i, j, k, kk, mm;
char  *s;
unsigned char buf[BUFSIZE];

/* open the image file */
fl = open(IPfile,0);

/* determine various image parameters from raster file header */
for (i=1; i<=8; i++)
    {
    nr = read(fl,buf,4);
    length = buf[0]*256*256*256+buf[1]*256*256+buf[2]*256+buf[3];
    switch (i)
```

```
            {
            case 1: ras_magic = length;
                    break;
            case 2: ras_width = length;
                    break;
            case 3: ras_height = length;
                    break;
            case 4: ras_depth = length;
                    break;
            case 5: ras_length = length;
                    break;
            case 6: ras_type = length;
                    break;
            case 7: ras_maptype = length;
                    break;
            case 8: ras_maplength = length;
                    break;
            }
    }

/* skip colormap */
if (ras_maplength > 0)
    lseek(f1,ras_maplength,1);

/* read image data from file into 2-D array */
i = 0;
kk = 512 / ras_width;
while ((nr = read(f1,buf,512)) > 0)
      {
      mm = 0;
      for (k=1; k <= kk; k++)
          {
          for (j=0; j < ras_width; j++, mm++)
              Timage[i][j] = buf[mm];
          i++;
          }
      }

/* close image file */
close(f1);
}

/* ******************************************************* */
/* WRITE THE PROCESSED IMAGE TO A FILE IN RASTER FORMAT */
/* ******************************************************* */
char *OPfilename;
writeimage(OPfilename)
{
int   f2, length, i, j, n1, n2, n3;
unsigned char buf2[BUFSIZE];

/* create a file to store the processed image */
f2 = creat(OPfilename,0644);

/* create header for raster file */
n1 = 256;
n2 = 256 * 256;
n3 = 256 * 256 *256;
for (i=1; i<=8; i++)
    {
    switch (i)
           {
           case 1: length = ras_magic;
                   break;
           case 2: length = ras_width;
                   break;
```



```
           case 3: length = ras_height;
                   break;
           case 4: length = ras_depth;
                   break;
           case 5: length = ras_length;
                   break;
           case 6: length = ras_type;
                   break;
           case 7: if (ras_maptype == 2)
                   /* checkboard and crossboard images */
                       length = 1;
                   else
                       length = ras_maptype;
                   break;
           case 8: if (ras_maplength == 0)
                   /* checkboard and crossboard images */
                       length = 768;
                   else
                       length = ras_maplength;
                   break;
           }
    buf2[0] = length / n3;
    buf2[1] = (length - buf2[0]*n3) / n2;
    buf2[2] = (length - buf2[0]*n3 - buf2[1]*n2) / n1;
    buf2[3] = (length - buf2[0]*n3 - buf2[1]*n2 - buf2[2]*n1);
    write(f2,buf2,4);
    }

/* create the colourmap for monochrome image */
for (i=0; i < 256; i++)
    buf2[i] = i;
write(f2, buf2, 256);   /* red */
write(f2, buf2, 256);   /* green */
write(f2, buf2, 256);   /* blue */

/* write enhanced images into files */
for (i=0; i < ras_height; i++)
    {
    for (j=0; j < ras_width; j++)
        buf2[j] = Map[i][j];
    write(f2,buf2,ras_width);
    }

close(f2);
}

/* ************************************************************************ */
/* A MATRIX MUST BE DECOMPOSED INTO ALPHA AND BETA BEFORE FINDING INVERSE */
/*     THIS PROCEDURE IS TO FIND THE INVERSE OF ARRAY DEFINITED AS d      */
/* ************************************************************************ */
Inver_D()
{
LuDcmp();
Inverse();
}

/* ********************************************* */
/* TO DECOMPOSE A MATRIX TO TWO TRIANGULAR MATRIX */
/*   ONE IS CALLED ALPHA AND ONE IS CALLED BETA   */
/* ********************************************* */
LuDcmp()
{
int   i,j,k;
float temp;

for (j=0;j<V;j++)
```

```
    alpha[j][j]=1;
for (j=0;j<V;j++)
    {
    for (i=0;i<=j;i++)
        {
        temp=0;
        for (k=0;k<i;k++)
            temp+=(alpha[i][k]*beta[k][j]);
        beta[i][j]=d[i][j]-temp;
        }
        if (j < (V-1))
           {
           for (i=j+1;i<V;i++)
               {
               temp=0;
               for (k=0;k<j;k++)
                   temp+=(alpha[i][k]*beta[k][j]);
               if (d[i][j] == temp)
                  alpha[i][j]=0;
               else
                  alpha[i][j]=(d[i][j]-temp)/beta[j][j];
               }
           }
    }
}

/* ************************************************************ */
/* FIND THE INVERSE OF A MATRIX WHICH HAS BEEN DECOMPOSED ALREADY */
/* ************************************************************ */
Inverse()
{
int   row,col,num;
float temp[NumPoint+2][NumPoint+2];

for (col=0;col<V;col++)
    {
    for (num=0;num<V;num++)
        f[num]=0;
    f[col]=1;
    LuFwSb();
    LuBkSb();
    for (num=0;num<V;num++)
        temp[num][col]=f[num];
    }
for (row=0;row<V;row++)
    for (col=0;col<V;col++)
        d[row][col]=temp[row][col];
}

/* ********************** */
/* LU FORWARD SUBSTITUTION */
/* ********************** */
LuFwSb()
{
int   i,j;
float temp;
float y[NumPoint+2];

for (i=0;i<V;i++)
    {
    temp=0;
    for (j=0;j<i;j++)
        temp+=(alpha[i][j]*y[j]);
    y[i]=(f[i]-temp)/alpha[i][i];
    }
for (i=0;i<V;i++)
```



```
    f[i]=y[i];
}

/* ************************ */
/* LU BACKWARD SUBSTITUTION */
/* ************************ */
LuBkSb()
{
int   i,j;
float temp;
float y[NumPoint+2];

for (i=V-1;i>=0;i--)
    {
    temp=0;
    for (j=V-1;j>i;j--)
        temp+=(beta[i][j]*y[j]);
    if (f[i] == temp)
       y[i]=0;
    else
       y[i]=(f[i]-temp)/beta[i][i];
    }
for (i=0;i<V;i++)
    f[i]=y[i];
}

/* ************************************************************ */
/* THIS PROCEDURE IS USED FOR DISPLAYING THE MASK OF A0,A2,d,f */
/* e.g. 1,0,1,0 : Display A0 and d only!!!                      */
/* ************************************************************ */
TestMsk(num1,num2,num3,num4)
{
int   row,col;

/*   if (V == 0)
    max=NumPoint;
else
    max=V;
*/
printf("\n");
if (num1 == 1)
    {
    printf("The array of A0\n");
    printf("~~~~~~~~~~~~~~~\n");
    for (row=0;row<M;row++)
        {
        for (col=0;col<V;col++)
            printf("%8.3f ",A0[row][col]);
        printf("\n");
        }
    }
if (num2 == 1)
    {
    printf("The array of A2\n");
    printf("~~~~~~~~~~~~~~~\n");
    for (row=0;row<M;row++)
        {
        for (col=0;col<V;col++)
            printf("%8.3f ",A2[row][col]);
        printf("\n");
        }
    }
if (num3 == 1)
    {
    printf("The array of d\n");
    printf("~~~~~~~~~~~~~~\n");
```

```
    for (row=0;row<V;row++)
        {
        for (col=0;col<V;col++)
            printf("%8.3f ",d[row][col]);
        printf("\n");
        }
    }
if (num4 == 1)
    {
    printf("The array of f\n");
    printf("~~~~~~~~~~~~~~\n");
    for (row=0;row<V;row++)
        printf("%8.3f ",f[row]);
    printf("\n");
    }
}

CUproc()
{

Calcu_D();
Inver_D();
Calcu_M();
}

Initialize()
{
BasicFcts();
TestMsk(1,1);
Calcu_A();
GenMask();
SetRange();
Renewimage();

}

/* ********************** */
/* EDGE DETECTION PROCEDURE */
/* ********************** */
Test()
{
int   row,col,mid;
int   CntDat;

CUproc();

mid=M/2+Pjsize/2;
for (row=mid;row<ras_height-mid;row++)
    for (col=mid;col<ras_width-mid;col++)
        {
          if (CalGrad(row,col)==1)
            {
              CalData(row,col);
              GetCoef();
              testing(row,col);
            }
        };
};

/* testing ********************************* */

testing(r,c)
{
int   mid;
float S1,S2,S2L,S2R;
```



```
mid=V/2;


S1=(f[mid+1]-f[mid-1])/2;

if (fabs(S1)>=threshold )
  {
    S2=f[mid-1]-2*f[mid]+f[mid+1];
    S2R=f[mid]+f[mid+2]-2*f[mid+1];
    if (S2*S2R<=0 && fabs(S2R)>=fabs(S2))
      Map[r][c]=TRUE;
    else
      { S2L=f[mid-2]+f[mid]-2*f[mid-1];
        if (S2*S2L<=0 && fabs(S2L)>=fabs(S2))
          Map[r][c]=TRUE;
      };
  };
};


Usage()
{

fprintf (stderr, "Command:\n");
fprintf (stderr, "%s InputFile, OutFile, scale, Pjsize, NomFitting, threshold,\n", Pr
ogName);

}

Abort(string)
char *string;
{

if (*string)
fprintf (stderr, "%s ERROR: %s\n", ProgName, string);

exit(3);
}



/* ************************************************************ */

Renewimage()
{
int i,j;


    for(i=0;i<ras_height;i++)
      for(j=0;j<ras_width;j++)
        Map[i][j]=FALSE;

}
```

Appendix D

The codes of Texture Focusing and for the computation of PCS.

```
/* ************************************************************************ */
/* Name of the programme: tf.c                                              */
/*                                                                          */
/* Function: The Texture Focusing texture segmentation scheme               */
/* ************************************************************************ */
/* To Compile this source code: make tf                                     */
/*                                                                          */
/* To run the programme:                                                    */
/* tf input output robust_margin no_level no_fixmap                         */
/*                                                                          */
/* where                                                                    */
/* tf: the name of this executable programme.                               */
/* input: the file name for the input image.                                */
/* output: the file name for the output image in the raster format.         */
/* robust_margin: defined in the thesis (Chapter 5)                         */
/* no_level: the number of levels in the multiresolution process            */
/* no_fixmap: a binary value. If no_fixmap=0, then produce fix map.         */
/* ************************************************************************ */
#include <stdio.h>
#include <stdlib.h>
#include "rasterio.h"
#include <math.h>
#include "stats.h"
#include "mts.h"

#define MAXIMAGE 513
#define No_Feature 40
#define No_Boundary 8
#define No_B2 5
#define No_B3 3


char            ProgName[20];

int nofeature;

void Usage()
{
    fprintf (stderr, "%s: input output robust_margin no_level no_fixmap\n", ProgName)
;
}                               /* Usage */


void main(int argc, char *argv[])
{
unsigned char   *IBuff,*ibuff;          /* image buffer                 */
unsigned char   *OBuff,*obuff;          /* image buffer                 */

int Row, rowext,                        /* # rows in image              */
    Col, colext;                        /* # columns in image           */

int fd1, fd2, fd3;

char ss[15];

float arand, prob;
struct feature afeature;
struct feature *head, *temp, *temp2;
struct class_table class_temp;
struct class_table class[MAXIMAGE][MAXIMAGE];
float mean[MAXIMAGE][MAXIMAGE];
float std[MAXIMAGE][MAXIMAGE];
float skew[MAXIMAGE][MAXIMAGE];
struct feature *femar[No_Boundary];
int nomar[No_Boundary];
```



```
float amean, astd, askew;
float robust_margin, margin, margin_step, coor;
int nofeat, nolevel, level, wlevel, nonode, arow, no;
int i, j, k, kk, hi, hj, bon, ratio, value;
int Recheck, boolean, nonomax, nonomar;
float incre, incre2, incre3, increment;
float incre_level, incre_level2, incre_level3, dist;
int ratio_incre, acount;
int no_fixed;
float ratio_fixed;
int no_fixmap;

strcpy (ProgName, argv[0]);

if ((argc < 6) || (argc > 6)) {
  Usage ();
  Abort ("");
}

if ((fd1 = open(argv[1], 0644)) == -1)
  Abort ("could not open input file");

robust_margin = atof (argv[3]);
nolevel = atoi (argv[4]);
nofeat = No_Feature;
no_fixmap = atoi (argv[5]);
if ((no_fixmap!=0) && (no_fixmap !=1))
  Abort ("no_fixmap=0 : produce fix map");

/* read in initial image header */
rhead(fd1, &Row, &Col);

/* alloc buffer memory */
IBuff = (unsigned char *) malloc (Row * Col * sizeof (char));
OBuff = (unsigned char *) malloc (Row * Col * sizeof (char));
if ((IBuff == NULL) || (OBuff == NULL))
  Abort ("Cannot allocate image buffer memory");

/* process images */

ibuff=IBuff;
FRead(fd1,ibuff,Row*Col);
head=&afeature;

ibuff=IBuff;
mean[0][0]=image_mean(ibuff,Row,Col);
if (DEBUG) printf("Mean=%f\n",mean[0][0]);

ibuff=IBuff;
std[0][0]=sqrt(image_var(ibuff,Row,Col,mean[0][0]));

ibuff=IBuff;
skew[0][0]=cbrt(image_skew(ibuff,Row,Col,mean[0][0]));

afeature.mean=mean[0][0];
afeature.std=std[0][0];
afeature.skew=skew[0][0];
afeature.no=1;
afeature.link=NULL;
class[0][0].link=&afeature;
class[0][0].fixed=0;
nofeature=1;

margin_step = robust_margin*distance(0.0, 0.0, 0.0, class[0][0].link);
margin=margin_step;
```

```
incre=incre2=incre3=0;
if (nolevel>2)
  {
    incre=(No_Boundary*1.0)/(nolevel-2);
    incre2=(No_B2*1.0)/(nolevel-2);
    incre3=(No_B3*1.0)/(nolevel-2);
  }

for (level=1;level<=nolevel;level++)
{

  /* SPLIT */
  /* propagate classes */

  if (1) printf("\nLevel=%d\n",level);
  bon=power(2,level);
  if (DEBUG) printf("Bon=%d\n",bon);
  for (i=bon-1;i>=0;i--)
    {
      for (j=bon-1;j>=0;j--)
        {
          hi=i/2;
          if (DEBUG) printf("Hi=%d\n",hi);

          hj=j/2;
          if (DEBUG) printf("Hj=%d\n",hj);

          class_temp=class[hi][hj];
          if (DEBUG) printf("Tmean=%f\n",temp->mean);
          class[i][j]=class_temp;
          if (DEBUG) printf("Cmean=%f\n",class[i][j].link->mean);
        };
    };

  if (0) printf("Quadraple Feature\n");

  temp=head;

  quadruple_feature(temp);

  if (0) printf("Process Sub-Image\n");

  if (0) printf ("Incre_Level=%f\n",incre_level);
  rowext=Row/bon;
  colext=Col/bon;

  for (i=0;i<bon;i++)
    {
      for (j=0;j<bon;j++)
        {
          if (class[i][j].fixed==0)
            {
              ibuff=IBuff;
              mean[i][j]=
                getmean(ibuff,i*rowext,j*colext,(i+1)*rowext-1,(j+1)*colext-1,Row);
              if (DEBUG) printf("Sub-Mean=%f\n",mean[i][j]);
              ibuff=IBuff;
              std[i][j]=sqrt(variance(ibuff,i*rowext,j*colext,
                     (i+1)*rowext-1,(j+1)*colext-1,Row,mean[i][j]));
              skew[i][j]=cbrt(skewness(ibuff,i*rowext,j*colext,
                       (i+1)*rowext-1,(j+1)*colext-1,Row,mean[i][j]));
              if (DEBUG) printf("Sub-Std=%f\n",std[i][j]);
              if (DEBUG) printf("Sub-Skew=%f\n",skew[i][j]);
              Recheck=0;
              dist=distance(mean[i][j],std[i][j],skew[i][j],class[i][j].link);
              if (dist==0)
```



```
                Recheck=0;
              else if (dist>=margin)
                Recheck=1;
              else
                {
                  prob=(margin-dist)/margin;
                  if (0) printf ("Prob=%f\n",prob);
                  if (prob<0)
                    Abort ("Probability must be greater than zero.");
                  else if (nn_rand()>prob) Recheck=1;
                };
              if (Recheck)
                {
                  (class[i][j].link->no)--;
                  class[i][j].link=
                    checkfeature(mean[i][j],std[i][j],skew[i][j],head,nofeat,margin);
                  (class[i][j].link->no)++;
                };
            };
        };
    };

  margin=margin+margin_step;

  /* Fix */
  if ((level>2))
    {
      incre_level=incre*(level-2);
      incre_level2=incre2*(level-2);
      incre_level3=incre3*(level-2);

      if (0) printf ("Incre_Level=%f\n",incre_level);
      if (0) printf ("Incre_Level2=%f\n",incre_level2);

      /* central area */
      for (i=1; i<=bon-2;i++)
        for(j=1; j<=bon-2; j++)
          {
            no=0;
            if (class[i-1][j-1].link==class[i][j].link) no++;
            if (class[i-1][j].link==class[i][j].link) no++;
            if (class[i-1][j+1].link==class[i][j].link) no++;
            if (class[i][j-1].link==class[i][j].link) no++;
            if (class[i][j+1].link==class[i][j].link) no++;
            if (class[i+1][j-1].link==class[i][j].link) no++;
            if (class[i+1][j].link==class[i][j].link) no++;
            if (class[i+1][j+1].link==class[i][j].link) no++;

            prob=(no*1.0+incre_level-No_Boundary*1.0)/(incre_level);
            if (0) printf("Prob=%f\n",prob);

            if (prob>1)
              Abort ("Wrong Probability.");
            else if (prob>0)
              {
                if (0) printf("Prob=%f\n",prob);
                if (nn_rand()<=prob)
                  class[i][j].fixed=1;
              };
          };
      /* top border */
      for(j=1; j<=bon-2; j++)
        {
          no=0;
          if (class[0][j-1].link==class[0][j].link) no++;
          if (class[1][j-1].link==class[0][j].link) no++;
```

```
      if (class[1][j].link==class[0][j].link) no++;
      if (class[1][j+1].link==class[0][j].link) no++;
      if (class[0][j+1].link==class[0][j].link) no++;

      prob=(no*1.0+incre_level2-No_B2*1.0)/(incre_level2);
      if (0) printf("Prob=%f\n",prob);

      if (prob>1)
        Abort ("Wrong Probability.");
      else if (prob>0)
        {
          if (0) printf("Prob=%f\n",prob);
          if (nn_rand()<=prob)
            class[0][j].fixed=1;
        };
    };
  /* bottom border */
  for(j=1; j<=bon-2; j++)
    {
      no=0;
      if (class[bon-1][j-1].link==class[bon-1][j].link) no++;
      if (class[bon-2][j-1].link==class[bon-1][j].link) no++;
      if (class[bon-2][j].link==class[bon-1][j].link) no++;
      if (class[bon-2][j+1].link==class[bon-1][j].link) no++;
      if (class[bon-1][j+1].link==class[bon-1][j].link) no++;

      prob=(no*1.0+incre_level2-No_B2*1.0)/(incre_level2);
      if (prob>1)
        Abort ("Wrong Probability.");
      else if (prob>0)
        {
          if (0) printf("Prob=%f\n",prob);
          if (nn_rand()<=prob)
            class[bon-1][j].fixed=1;
        };
    };
  /* left borer */
  for(i=1; i<=bon-2; i++)
    {
      no=0;
      if (class[i-1][0].link==class[i][0].link) no++;
      if (class[i-1][1].link==class[i][0].link) no++;
      if (class[i][1].link==class[i][0].link) no++;
      if (class[i+1][1].link==class[i][0].link) no++;
      if (class[i+1][0].link==class[i][0].link) no++;

        prob=(no*1.0+incre_level2-No_B2*1.0)/(incre_level2);
        if (prob>1)
          Abort ("Wrong Probability.");
        else if (prob>0)
          {
            if (0) printf("Prob=%f\n",prob);
            if (nn_rand()<=prob)
              class[i][0].fixed=1;
          };
    };
  /* right border */
  for(i=1; i<=bon-2; i++)
    {
      no=0;
      if (class[i-1][bon-1].link==class[i][bon-1].link) no++;
      if (class[i-1][bon-2].link==class[i][bon-1].link) no++;
      if (class[i][bon-2].link==class[i][bon-1].link) no++;
      if (class[i+1][bon-2].link==class[i][bon-1].link) no++;
      if (class[i+1][bon-1].link==class[i][bon-1].link) no++;
```



```
      prob=(no*1.0+incre_level2-No_B2*1.0)/(incre_level2);
      if (prob>1)
        Abort ("Wrong Probability.");
      else if (prob>0)
        {
          if (0) printf("Prob=%f\n",prob);
          if (nn_rand()<=prob)
            class[i][bon-1].fixed=1;
        };
    };

  /* upper left */
  no=0;
  if (class[0][1].link==class[0][0].link) no++;
  if (class[1][1].link==class[0][0].link) no++;
  if (class[1][0].link==class[0][0].link) no++;
  prob=(no*1.0+incre_level3-No_B3*1.0)/(incre_level3);
  if (prob>1)
    Abort ("Wrong Probability.");
  else if (prob>0)
    {
      if (0) printf("Prob=%f\n",prob);
      if (nn_rand()<=prob)
        class[0][0].fixed=1;
    };
  /* upper right */
  no=0;
  if (class[0][bon-2].link==class[0][bon-1].link) no++;
  if (class[1][bon-2].link==class[0][bon-1].link) no++;
  if (class[1][bon-1].link==class[0][bon-1].link) no++;
  prob=(no*1.0+incre_level3-No_B3*1.0)/(incre_level3);
  if (prob>1)
    Abort ("Wrong Probability.");
  else if (prob>0)
    {
      if (0) printf("Prob=%f\n",prob);
      if (nn_rand()<=prob)
        class[0][bon-1].fixed=1;
    };
  /* lower left */
  no=0;
  if (class[bon-2][0].link==class[bon-1][0].link) no++;
  if (class[bon-2][1].link==class[bon-1][0].link) no++;
  if (class[bon-1][1].link==class[bon-1][0].link) no++;
  prob=(no*1.0+incre_level3-No_B3*1.0)/(incre_level3);
  if (prob>1)
    Abort ("Wrong Probability.");
  else if (prob>0)
    {
      if (0) printf("Prob=%f\n",prob);
      if (nn_rand()<=prob)
        class[bon-1][0].fixed=1;
    };
  /* lower right */
  no=0;
  if (class[bon-2][bon-1].link==class[bon-1][bon-1].link) no++;
  if (class[bon-2][bon-2].link==class[bon-1][bon-1].link) no++;
  if (class[bon-1][bon-2].link==class[bon-1][bon-1].link) no++;
  prob=(no*1.0+incre_level3-No_B3*1.0)/(incre_level3);
  if (prob>1)
    Abort ("Wrong Probability.");
  else if (prob>0)
    {
      if (0) printf("Prob=%f\n",prob);
      if (nn_rand()<=prob)
        class[bon-1][bon-1].fixed=1;
```

```
        };

    };
  if (DEBUG) printf("Tidy Feature\n");

  head=tidyfeature(head);

  if (DEBUG) printf("Adjust feature statistics\n");
  temp=head;
  for (k=0; k<nofeature; k++)
    {
      nonode=temp->no;
      acount=nonode;
      amean=0;
      askew=0;
      astd=0;
      for (i=0;i<bon;i++)
        for (j=0;j<bon;j++)
          {
            if (class[i][j].link==temp)
              {
                amean=amean+mean[i][j];
                askew=askew+skew[i][j];
                acount--;
                if (acount<0) Abort ("Node number too small.");
              };
          };
      if (acount>0) Abort ("Node number too large.");
      amean=amean/nonode;
      askew=askew/nonode;

      for (i=0;i<bon;i++)
        for (j=0;j<bon;j++)
          {
            if (class[i][j].link==temp)
              {
                astd=std[i][j]*std[i][j]+(amean-mean[i][j])*(amean-mean[i][j]);
              };
          };

      temp->mean=amean;
      temp->skew=askew;
      temp->std=sqrt(astd/nonode);
      temp=temp->link;
    };

  no_fixed=0;
  for (i=0;i<bon;i++)
    for (j=0;j<bon;j++)
      {
        if (class[i][j].fixed==1)
          no_fixed++;
      };
  ratio_fixed=((float) no_fixed)/(bon*bon);
  if (1) printf("Ratio_Fixed=%f", ratio_fixed);
};


temp=head;
if (DEBUG) printf("Assign Colour\n");
assigncolour(temp);

if (DEBUG) printf("Re-Size\n");
wlevel=1;
arow=2;
```



```
do {
arow=arow*2;
wlevel++;
}
while(arow<Row);

ratio=power(2,(wlevel-nolevel));
if (0) printf("Ratio=%d\n", ratio);

for (i=Row-1; i>=0; i--)
{
  for (j=Col-1; j>=0; j--)
    {
      hi=i/ratio;
      hj=j/ratio;
      class_temp=class[hi][hj];
      class[i][j]=class_temp;
    };
};

if (0) printf("Produce Segmentation Map\n");

obuff=OBuff;
for (i=0; i<Row; i++)
{
  for (j=0; j<Col; j++)
    {
      value=class[i][j].link->colour;
      *obuff = (unsigned char) value;
      obuff++;
    }
}

if (0) printf("Output\n");


if ((fd2 = creat(argv[2], 0644)) == -1)
  Abort ("could not create output file");

whead(fd2, Row, Col);

obuff=OBuff;
FWrite(fd2, obuff, Row*Col);

close(fd1);
close(fd2);

if (no_fixmap==0)
{
  obuff=OBuff;
  for (i=0; i<Row; i++)
    {
      for (j=0; j<Col; j++)
        {
          value=(class[i][j].fixed)*200;
          *obuff = (unsigned char) value;
          obuff++;
        }
    }

  strcpy(ss,argv[2]);
  strcat(ss,".fix");
  if ((fd3 = creat(ss, 0644)) == -1)
    Abort ("could not create fixed map");

  whead(fd3, Row, Col);
```

```
   obuff=OBuff;
   FWrite(fd3, obuff, Row*Col);

   close(fd3);
 }
 Abort ("");
 }                                   /* main */
```

```
/* ************************************************************************ */
/* Name of the programme: pcs.c                                             */
/*                                                                          */
/* Function: To calculate the percentage of the correct segmentation (PCS)  */
/* ************************************************************************ */
/* To Compile this source code: make pcs                                    */
/*                                                                          */
/* To run the programme:                                                    */
/* pcs file_name radius BG-colour CTR-colour                                */
/*                                                                          */
/* where                                                                    */
/* pcs: the name of this executable programme.                              */
/* file_name: the file name for the input image.                            */
/* BG-colour: the grey-level of the background.                             */
/* CTR-colour: the grey-level of the  central area                          */
/* ************************************************************************ */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "rasterio.h"

#define ImagSize   512
#define BUFSIZE    512

char  ProgName[15];

unsigned char   Image[ImagSize][ImagSize];

int Row1,Col1;

float  ratio();

void Usage()
{
fprintf (stderr, "Command: %s file_name radius BG-colour CTR-colour\n", ProgName);
}

void main(argc,argv)
int  argc;
char *argv[];
{

int fd1;
FILE *fid;
int i,j,mm;
unsigned char buf[BUFSIZE];
unsigned char *ptr;
char  filename[15];
float pcs_value;

strcpy (ProgName, argv[0]);
if (argc != 5)
  {
    Usage();
    Abort("");
  }

strcpy (filename, argv[1]);
if ((fd1 = open(argv[1], 0644)) == -1)
  Abort ("could not open input file.");

/* read in initial image header */
rhead(fd1, &Row1, &Col1);

for (i=0; i<Row1;i++)
  {
```



```
    FRead(fd1,buf,Col1);
    mm=0;
    for (j=0; j < Col1; j++, mm++)
      Image[i][j] = buf[mm];
  };

pcs_value=ratio(atoi(argv[2]),atoi(argv[3]),atoi(argv[4]));
printf ("file=%s\tPCS=%4.2f\n",filename,pcs_value);

close(fd1);

fid=fopen("datafile","a");
fprintf (fid,"file=%s\tPCS=%4.2f\n",filename,pcs_value);
fclose(fid);

Abort ("");
}                                /* main */

float ratio(int rad, int bg_colour, int fg_colour)
{
int x_centre, y_centre, xx, yy, rr;
int rad2;
int i, j;
int count;

/* determine centre of mask before any enlargement if odd-size image */
xx = Row1;
yy = Col1;
x_centre = xx / 2;
y_centre = yy / 2;

count=0;
rad2=rad*rad;
for (i=0; i<Col1; i++)
  {
    for (j=0; j < Row1; j++)
      { rr=(i-y_centre)*(i-y_centre)
          +(j-x_centre)*(j-x_centre);
      if (rr<rad2)
        {
          if (Image[i][j]==fg_colour) count++;
        }
      else
        if (Image[i][j]==bg_colour) count++;

      }
  }
if (1) printf("count=%d", count);
return((1.0*count)/(Row1*Col1));
};
```

```
struct feature {
float mean;
float std;
float skew;
int no;
int colour;
struct feature *link;
};

struct class_table {
struct feature *link;
int fixed;
};

extern int nofeature;


#ifndef _MTS_H
#define _MTS_H


void assigncolour(struct feature *pointer);
void quadruple_feature(struct feature *pointer);
struct feature *checkfeature(float amean, float astd, float askew, struct feature *fh
ead, int nofeat, float margin);
float distance(float amean, float astd, float askew, struct feature *pointer);
struct feature *tidyfeature(struct feature *ptr);
#endif
```

```
struct feature {
float mean;
float std;
float skew;
int no;
int colour;
struct feature *link;
};

struct class_table {
struct feature *link;
int fixed;
};

extern int nofeature;


#ifndef _MTS_H
#define _MTS_H


void assigncolour(struct feature *pointer);
void quadruple_feature(struct feature *pointer);
struct feature *checkfeature(float amean, float astd, float askew, struct feature *fh
ead, int nofeat, float margin);
float distance(float amean, float astd, float askew, struct feature *pointer);
struct feature *tidyfeature(struct feature *ptr);
#endif
```

```
#include <stdio.h>
#include "mts.h"
#include "rasterio.h"



void assigncolour(struct feature *pointer)
{

struct feature *ptr, *temp;
int colour, cjump, cjump2, iter;

cjump=255/nofeature;
cjump2=cjump/3;

iter=nofeature;
ptr=pointer;
temp=ptr;
colour=0;

do {
colour=colour+2*cjump2;
if (0) printf("Colour=%d\n",colour);
temp->colour=colour;
temp=(temp->link);
iter--;
if (iter<0) Abort ("Feature table is too long (assign colour).");
} while(temp!=NULL);

if (iter>0) Abort ("Feature table is too short (assign colour).");
};



void quadruple_feature(struct feature *pointer)
{
struct feature *temp;
int iter;

iter=nofeature;
temp=pointer;
do {
temp->no=4*(temp->no);
if (0) printf("No=%d\n", temp->no);

temp=temp->link;
iter--;
if (iter<0) Abort ("Feature table is too long.");
} while(temp!=NULL);

if (iter>0) Abort ("Feature table is too short.");
};


struct feature *checkfeature(float amean, float astd, float askew, struct feature *fh
ead, int nofeat, float margin)
{
int iter, counter;
float adist, mindist;
struct feature *temp, *temp2, *temp3;

iter=nofeature;
temp=fhead;
counter=0;
mindist=100000000;
```



```
if (DEBUG) printf("No. of Feature=%d\n",nofeature);
do {
if (DEBUG) printf ("Sub_Mean=%f\n",amean);
if (DEBUG) printf ("Sub_Std=%f\n",astd);
if (DEBUG) printf ("Feature_Mean=%f\n",temp->mean);
if (DEBUG) printf ("Feature_Std=%f\n",temp->std);

adist=distance(amean, astd, askew, temp);
if (DEBUG) printf ("Dist=%f\n",adist);
if (adist<margin)
  {
    counter++;
    if (adist < mindist)
      {
        temp2=temp;
        mindist=adist;
      }
    else
      {
        if (DEBUG) printf("Adist(%f) > Mindist(%f)\n", adist,mindist);
      };
  };
temp3=temp;
temp=temp->link;
iter--;
if (iter<0) Abort ("Feature table is too long (check feature).");
} while(temp!=NULL);

if (iter>0) Abort ("Feature table is too short (check feature).");

/* if (counter>7) Abort ("Margin too large."); */

if (counter==0)
  {
    temp2=(struct feature *) malloc (sizeof (struct feature));
    temp3->link=temp2;
    temp2->mean=amean;
    temp2->std=astd;
    temp2->skew=askew;
    temp2->no=0;
    temp2->link=NULL;
    nofeature++;
  };
if (nofeature>nofeat) Abort ("Two many features detected.");
return (temp2);
};


float distance(float amean, float astd, float askew, struct feature *pointer)
{
float adist;
float value, value2;
struct feature *ptr;

ptr=pointer;

value=astd-(ptr->std);
value2=askew-(ptr->skew);
adist= (float) sqrt((amean-(ptr->mean))*(amean-(ptr->mean))+value*value+value2*value2
);

return (adist);
};


struct feature *tidyfeature(struct feature *ptr)
```

```
{
int iter;
struct feature *ahead, *temp, *temp2;

temp=ptr;
while (temp->no<=0)
  {
    if (1) printf("Feature head cut off\n");
    if (0) printf("No=%d\n", temp->no);
    if (temp->no<0) Abort ("Wrong node number.");
    temp=temp->link;
    nofeature--;
  };

ahead=temp;
if (0) printf("No=%d\n", temp->no);

temp2=temp->link;
while(temp2!=NULL)
{
  if (0) printf("No=%d\n", temp2->no);

  if (temp2->no<=0)
    {
      if (1) printf("Feature table cut off\n");
      if (temp->no<0) Abort ("Wrong node number.");

      temp->link=temp2->link;
      temp2=temp2->link;
      nofeature--;
    }
  else
    {
      temp=temp2;
      temp2=temp2->link;
    };
};
if (1) printf("No. of Feature=%d\n",nofeature);
return(ahead);

};
```

```
#ifndef _RASTERIO_H
#define _RASTERIO_H

void Abort(char   *string);
void whead(int fd, int rows, int cols);
void rhead(int fd, int *pRows, int *pCols);
void rhead(int fd, int *pRows, int *pCols);
void outint(int fd, int m);
void FRead(int fd, unsigned char *pointer, int numbytes);
void FWrite(int fd, unsigned char *pointer, int len);
void RScale(float *fbuff, unsigned char *cbuff, int numpix);

#endif

#define DEBUG 0
```

```
#include <stdio.h>
#include <math.h>
#include "rasterio.h"

void Abort(char *string)
{
    if (*string)
        fprintf (stderr, "ERROR: %s\n", string);

    exit (3);
}                                   /* Abort */

void whead(int fd, int rows, int cols)
{
  outint(fd,1504078485);
  outint(fd,cols);
  outint(fd,rows);
  outint(fd,8);
  outint(fd,cols*rows);
  outint(fd,1);
  outint(fd,2);
  outint(fd,0);
}

void rhead(int fd, int *pRows, int *pCols)
{
        unsigned char buf[32];
        int length;

        if (read(fd, buf, 32)!=32)
                abort("Read Image header error.");

        *pCols=buf[4]*256*256*256+buf[5]*256*256+buf[6]*256+buf[7];
        *pRows=buf[8]*256*256*256+buf[9]*256*256+buf[10]*256+buf[11];
        length=buf[28]*256*256*256+buf[29]*256*256+buf[30]*256+buf[31];
        if (1) printf("\nMaplength = %d\n", length);
        if (lseek (fd, length,1)==-1) Abort ("lseek error");
}

void outint(int fd, int m)
{ unsigned char buf[4];
  int n1,n2,n3,i;
  n1=256;
  n2=256*256;
  n3=256*256*256;
  buf[0]=m/n3;
  buf[1]=(m-buf[0]*n3)/n2;
  buf[2]=(m-buf[0]*n3-buf[1]*n2)/n1;
  buf[3]=m-buf[0]*n3-buf[1]*n2-buf[2]*n1;
  if (write(fd,buf,4)!=4) abort("writing error\n");
}

 /*
  * FRead - read 'numbytes' to 'pointer' and check for errors.
  */


void FRead(int fd, unsigned char *pointer, int numbytes)
{

  if (read (fd, pointer, numbytes) != numbytes)
    Abort ("reached end of input file");

}                                   /* FRead */
```



```
 /*
  * FWrite - write 'len' bytes from 'pointer' to stdout, checking for errors.
  */

void FWrite(int fd, unsigned char *pointer, int len)
{
    if (write (fd, pointer, len) != len)
        Abort ("cannot write to output");
}                                   /* FWrite */


 /*
  * RScale: Re-scale floating buffer so it is between 0 -- 255.
  */

void RScale(float *fbuff, unsigned char *cbuff, int numpix)
{
  int n;
  float   max = 0.0, val, value;
  float scale;
  float *fbuffptr;
  unsigned char *cbuffptr;

  n=numpix;
  fbuffptr=fbuff;
  do {
    val = *(fbuffptr++);
    if (val > max)
      max = val;
  } while (--n);
  if (1) printf("Max=%f\n",max);
  scale =255/max;
  n=numpix;
  fbuffptr=fbuff;
  cbuffptr=cbuff;

  do
    {
      value = (*(fbuffptr++) * scale);
      if(value>255)
        *(cbuffptr++)=255;
      else if (value<0)
        *(cbuffptr++)=0;
      else
        *(cbuffptr++)=(unsigned char)value;
    }
  while (--n);

}
```

```
#include <sys/time.h>

#define MAXINT  0x7FFF
#define FALSE 0
#define TRUE 1

static char     RandFlag = FALSE;       /* true if 2nd rnd # generated  */

static float    Xx1,                    /* used by the 'NRand' routine  */
                Xx2;

static struct timeval   tv;
static struct timezone  tz;

#ifndef _STATES_H
#define _STATES_H

float image_mean(unsigned char *imageptr, int sizex, int sizey);
float image_var(unsigned char *imageptr, int sizex, int sizey, float mean);
float image_skew(unsigned char *imageptr, int sizex, int sizey, float mean);
float getmean(unsigned char *dataptr, int startx,int starty,int endx,int endy, int si
zex);
float variance (unsigned char *dataptr,int startx,int starty,int endx,int endy, int s
izex, float mean);
float skewness (unsigned char *dataptr,int startx,int starty,int endx,int endy, int s
izex, float mean);
void test(unsigned char *imageptr, int sizex, int sizey);
int power( int base, int n);
void gen_seed(void);
float n_rand(void);
float nn_rand(void);

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "rasterio.h"
#include "stats.h"

int power( int base, int n)
{
int i,p;

p=1;
for(i=1;i<=n;i++)
  p=p*base;

return p;
}


float image_mean(unsigned char *imageptr, int sizex, int sizey )
{
        float mean;
        int i,j;

        mean = 0;

        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
              mean = mean + *imageptr;
              imageptr++;
            };
        mean = mean / (sizex*sizey);

        if (DEBUG) printf("Image mean : %f\n", mean);

        return(mean);
}

float image_var(unsigned char *imageptr, int sizex, int sizey, float mean)
{
        float sum, var;
        int i,j;

        var = 0; sum = 0;
        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
              sum = sum + (*imageptr-mean)*(*imageptr-mean);
              imageptr++;
            };
        var = sum / (sizex*sizey);

        if (DEBUG) printf("Image variance : %f\n", var);

        return(var);
}

float image_skew(unsigned char *imageptr, int sizex, int sizey, float mean)
{
        float sum, skew;
        int i,j;

        skew = 0; sum = 0;
        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
```



```
              sum = sum
                + (*imageptr-mean)*(*imageptr-mean)*(*imageptr-mean);
              imageptr++;
            };
        skew = sum/(sizex*sizey);

        if (DEBUG) printf("Image skew : %f\n", skew);

        return(skew);
}

float getmean(unsigned char *dataptr, int startx,int starty,int endx,int endy, int si
zex)
{
        float mean;
        int i,j, mean2;
        unsigned char *ptr;

        mean=0.0;
        mean2=0;

        if (DEBUG) printf("value : %i\n", *dataptr);

        for (i=startx; i<=endx; i++)
              for (j=starty; j<=endy; j++)
                {
                  ptr=dataptr+(i*sizex+j);
                  if (0) printf("value : %i\n", (int)*ptr);
                  mean2 = mean2 + (int) *ptr;
                }
        if (DEBUG) printf("Image mean2 : %i\n", mean2);
        mean = (float) mean2/ (float)((endx-startx+1)*(endy-starty+1));
        if (DEBUG) printf("startx=%i, endx=%i, starty=%i, endy=%i\n", startx,endx,sta
rty,endy);

        if (DEBUG) printf("Image mean : %f\n", mean);

        return(mean);
}

float variance (unsigned char *dataptr,int startx,int starty,int endx,int endy, int s
izex, float mean)
{
        float var, sum, value;
        int i,j;
        unsigned char *ptr;
        if (DEBUG) printf(" %d %d %d %d \n", startx, starty, endx, endy);
        var = 0; sum =0;

        for (i=startx; i<=endx; i++)
          for (j=starty; j<=endy; j++)
            {
              ptr=dataptr+(i*sizex+j);
              value=(float)*ptr;
              sum = sum + (value - mean)*(value - mean)  ;
            }
        var = (float) sum / (float)((endx-startx+1) * (endy-starty+1) );

        return (var);
}

float skewness (unsigned char *dataptr,int startx,int starty,int endx,int endy, int s
izex, float mean)
{
        float skew, sum, value;
        int i,j;
```

```
        unsigned char *ptr;
        if (DEBUG) printf(" %d %d %d %d \n", startx, starty, endx, endy);
        skew = 0; sum =0;

        for (i=startx; i<=endx; i++)
          for (j=starty; j<=endy; j++)
            {
              ptr=dataptr+(i*sizex+j);
              value=(float)*ptr;
              sum = sum + (value - mean)*(value - mean)*(value - mean) ;
            }
        skew = (float) sum / (float)((endx-startx+1)*(endy-starty+1));

        return (skew);
}

void test(unsigned char *imageptr, int sizex, int sizey)
{
        int i,j;
        unsigned char *ptr;

        ptr=imageptr;
        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
              if (1) printf("Image value : %i\n", *ptr);
              ptr++;
            }
}

void test2(float *imageptr, int sizex, int sizey)
{
        int i,j;
        float *ptr;

        ptr=imageptr;
        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
              if (1) printf("Image value : %8.3f\n", *ptr);
              ptr++;
            }
}


void gen_seed(void)
{
  gettimeofday (&tv, &tz);      /* initialize random seed */
  srand (getpid() ^ tv.tv_sec ^ tv.tv_usec);
}

float n_rand(void)
{       float v1, v2, s, temp1, temp2;

        if (RandFlag) {  /* if RandFlag, then return 2nd random number */
                RandFlag = FALSE;
                if (0) printf("Rand2=%f\n",Xx2);
                return (Xx2);
        }
        RandFlag = TRUE;

       /*
        * The following code generates 2 Standard Normally distributed numbers
        * into 'Xx1' and 'Xx2' using the Polar method for normal deviates.
        */
```



```
        /*printf("start_rand\n");*/
        do {
            v1 = (double) (rand () & MAXINT) / (double) MAXINT * 2.0 - 1.0;
            v2 = (double) (rand () & MAXINT) / (double) MAXINT * 2.0 - 1.0;
            s = v1 * v1 + v2 * v2;
            /*printf("s=%f\n",s);*/

         } while (s >= 1.0);

        Xx1 = v1 * sqrt ((-2.0 * log (s)) / s);
        Xx2 = v2 * sqrt ((-2.0 * log (s)) / s);

        if (0) printf("Rand1=%f\n",Xx1);
        return (Xx1);
}

float nn_rand(void)
{
  float temp;

  do
    temp=n_rand();
  while ((temp<-0.5) ||(temp >0.5));

  temp=temp+0.5;
  return(temp);
}
```

Appendix E

The code for the motion field segmentation.

```
/* ************************************************************************ */
/* Name of the programme: of.c                                              */
/*                                                                          */
/* Function: The motion field segmentation scheme                           */
/* ************************************************************************ */
/* To Compile this source code: make of                                     */
/*                                                                          */
/* To run the programme:                                                    */
/* of current_frame previous_frame output margin fix_threshold no_fixmap    */
/*                                                                          */
/* where                                                                    */
/* of: the name of this executable programme.                               */
/* current_frame: the file name for the current frame of an image sequence  */
/* previous_frame: the file name for the previous frame of an image sequence*/
/* output: the file name for the output image in the raster format.         */
/* margin: defined in the thesis (Chapter 6)                                */
/* fix_threshold: the threshold for the fixing of a node in a quad-tree     */
/* no_fixmap: a binary value. If no_fixmap=0, then produce fix map.         */
/* ************************************************************************ */
#include <stdio.h>
#include <stdlib.h>
#include "rasterio.h"
#include <math.h>
#include "stats.h"
#include "mof.h"

#define MAXIMAGE 257
#define EXTREME 1000000000
#define No_Level 8
#define No_Feature 90
#define Start_Fix 0
#define StopLevel 0
#define Row256 256
#define Col256 256


char            ProgName[20];

int nofeature;
float u[MAXIMAGE][MAXIMAGE];
float v[MAXIMAGE][MAXIMAGE];
float error[MAXIMAGE][MAXIMAGE];
float space[MAXIMAGE][MAXIMAGE];
float ei[MAXIMAGE][MAXIMAGE];
float ej[MAXIMAGE][MAXIMAGE];
float et[MAXIMAGE][MAXIMAGE];

float get_uv(int i, int j);

float smooth_u(int i, int j, int bon);

float smooth_v(int i, int j, int bon);

void Usage()
{
    fprintf (stderr, "%s: current_frame previous_frame output margin fix_threshold no
_fixmap\n", ProgName);
}                               /* Usage */


void main(int argc, char *argv[])
{
unsigned char   *IBuff,*IBuff2,*ibuff,*ibuff2;
unsigned char   *OBuff,*obuff;          /* image buffer                 */

unsigned char buf[MAXIMAGE];
```



```
int Row,                        /* # rows in image              */
    Col;                        /* # columns in image           */

int fd1, fd2, fd3;

char ss[20];

float arand, prob;
struct feature afeature;
struct feature *head, *temp, *temp2;
struct class_table class_temp;
struct class_table class[MAXIMAGE][MAXIMAGE];

unsigned char image1[MAXIMAGE][MAXIMAGE];
unsigned char image2[MAXIMAGE][MAXIMAGE];

float margin, margin_step, fix_threshold;
int nofeat, level, wlevel, nonode, arow, no;
int i, j, k, kk, hi, hj, bon, ratio, value, last;
int Recheck, boolean, nonomax, nonomar;
float incre, incre2, incre3, increment;
float incre_level, incre_level2, incre_level3, dist;
int ratio_incre, acount;
int no_fixed;
float ratio_fixed;
int no_fixmap;
float u_ave, v_ave;

strcpy (ProgName, argv[0]);

if ((argc < 7) || (argc > 7)) {
  Usage ();
  Abort ("");
}

strcpy(ss, argv[1]);
strcat(ss, ".13");

if ((fd1 = open(ss, 0644)) == -1)
  Abort ("could not open first input file");

strcpy(ss, argv[2]);
strcat(ss, ".13");

if ((fd2 = open(ss, 0644)) == -1)
  Abort ("could not open second input file");

margin = atof (argv[4]);
nofeat = No_Feature;
fix_threshold=atof(argv[5]);
no_fixmap = atoi (argv[6]);
if ((no_fixmap!=0) && (no_fixmap !=1))
  Abort ("no_fixmap=0 : produce fix map");
last=0;

/* read in initial image header */
rhead(fd1, &Row, &Col);
rhead(fd2, &Row, &Col);

/* process images */

for(i=0; i<Row; i++)
  {
    FRead(fd1,buf,Col);
    for(j=0; j<Col; j++)
```

```
    {
      image1[i][j]=buf[j];
    }
  };
close(fd1);


for(i=0; i<Row; i++)
  {
    FRead(fd2,buf,Col);
    for(j=0; j<Col; j++)
    {
      image2[i][j]=buf[j];
    }
  };
close(fd2);

ratio=power(2,(No_Level-3));

for(i=0; i<Row; i++)
  {
    for(j=0; j<Col; j++)
    {
      et[i][j]=(float) (image2[i][j]-image1[i][j])/ratio; /* et contain -Et */
      if (j==0)
        ej[i][j]=(float) (image1[i][j+1]-image1[i][j])/ratio;
      else if (j==Col-1)
        ej[i][j]=(float) (image1[i][j]-image1[i][j-1])/ratio;
      else
        ej[i][j]=0.5*(image1[i][j+1]-image1[i][j-1])/ratio;

      if (i==0)
        ei[i][j]=(float) (image1[i+1][j]-image1[i][j])/ratio;
      else if (i==Row-1)
        ei[i][j]=(float) (image1[i][j]-image1[i-1][j])/ratio;
      else
        ei[i][j]=0.5*(image1[i+1][j]-image1[i-1][j])/ratio;
    };
  };

u_ave=0;
v_ave=0;

for(i=0; i<4; i++)      /* level 2 */
  for(j=0; j<4; j++)
    {
      error[i][j]=get_uv(i,j);
      u_ave=u_ave+u[i][j];
      v_ave=v_ave+v[i][j];
    };
u_ave=u_ave/16;
v_ave=v_ave/16;

head=&afeature;

afeature.u=u_ave;
afeature.v=v_ave;
afeature.no=16;
afeature.link=NULL;
nofeature=1;

margin_step = margin;

incre=incre2=incre3=0;

for(i=0; i<4; i++)
```



```
  for(j=0; j<4; j++)
    {
      Recheck=0;
      class[i][j].link=&afeature;
      class[i][j].fixed=0;

      dist=distance2(u[i][j],v[i][j],class[i][j].link);
      if (dist>0)
        {
          if (dist>=margin)
            Recheck=1;
          else
            {
              prob=(margin-dist)/margin;
              if (0) printf ("Prob=%f\n",prob);
              if (prob<0)
                Abort ("Probability must be greater than zero.");
              else if (nn_rand()>prob) Recheck=1;
            };
        };
      if (Recheck)
        {
          (class[i][j].link->no)--;
          class[i][j].link=
            checkfeature(u[i][j],v[i][j],head,nofeat,margin, last);
          (class[i][j].link->no)++;
        };
    };
  head=tidyfeature(head);

  if (DEBUG) printf("Adjust feature statistics\n");
  temp=head;
  for (k=0; k<nofeature; k++)
    {
      if (0) printf ("k=%d\n", k);
      nonode=temp->no;
      acount=nonode;
      u_ave=0;
      v_ave=0;
      for (i=0;i<4;i++)
        for (j=0;j<4;j++)
          {
            if (class[i][j].link==temp)
              {
                u_ave=u_ave+u[i][j];
                v_ave=v_ave+v[i][j];
                acount--;
                if (acount<0) Abort ("Node number too small.");
              };
          };
      if (acount>0) Abort ("Node number too large.");
      u_ave=u_ave/nonode;
      v_ave=v_ave/nonode;

      temp->u=u_ave;
      temp->v=v_ave;
      temp=temp->link;
    };

margin=margin+margin_step;

for (level=3;level<=(No_Level-1-StopLevel);level++)
{

  /* SPLIT */
  /* propagate classes */
```

```
if (1) printf("\nLevel=%d\n",level);
bon=power(2,level);
if (DEBUG) printf("Bon=%d\n",bon);
for (i=bon-1;i>=0;i--)
  {
    for (j=bon-1;j>=0;j--)
      {
        hi=i/2;
        if (DEBUG) printf("Hi=%d\n",hi);

        hj=j/2;
        if (DEBUG) printf("Hj=%d\n",hj);

        class_temp=class[hi][hj];
        class[i][j]=class_temp;
      };
  };
if (0) printf("Quadruple Feature\n");

temp=head;

quadruple_feature(temp);
if (0) printf("Process Sub-Image\n");

if (0) printf ("Incre_Level=%f\n",incre_level);

strcpy(ss, argv[1]);
if (level==3) strcat(ss, ".14");
if (level==4) strcat(ss, ".15");
if (level==5) strcat(ss, ".16");
if (level==6) strcat(ss, ".17");
if (level==7) strcat(ss, "");

if (0) printf ("%s\n",ss);

if ((fd1 = open(ss, 0644)) == -1)
  Abort ("could not open first input file");

rhead(fd1, &Row, &Col);

for(i=0; i<Row; i++)
  {
    FRead(fd1,buf,Col);
    for(j=0; j<Col; j++)
      {
        image1[i][j]=buf[j];
      }
  };
close(fd1);

strcpy(ss, argv[2]);
if (level==3) strcat(ss, ".14");
if (level==4) strcat(ss, ".15");
if (level==5) strcat(ss, ".16");
if (level==6) strcat(ss, ".17");
if (level==7) strcat(ss, "");

if ((fd2 = open(ss, 0644)) == -1)
  Abort ("could not open first input file");

rhead(fd2, &Row, &Col);

for(i=0; i<Row; i++)
  {
    FRead(fd2,buf,Col);
```



```
    for(j=0; j<Col; j++)
      {
        image2[i][j]=buf[j];
      }
  };
close(fd2);

ratio=power(2,(No_Level-level-1));
printf("ratio=%d", ratio);

for(i=0; i<Row; i++)
  {
    for(j=0; j<Col; j++)
      {
        et[i][j]=(float) (image2[i][j]-image1[i][j])/ratio; /* et contain -Et */
        if (j==0)
          ej[i][j]=(float) (image1[i][j+1]-image1[i][j])/ratio;
        else if (j==Col-1)
          ej[i][j]=(float) (image1[i][j]-image1[i][j-1])/ratio;
        else
          ej[i][j]=0.5*(image1[i][j+1]-image1[i][j-1])/ratio;

        if (i==0)
          ei[i][j]=(float) (image1[i+1][j]-image1[i][j])/ratio;
        else if (j==Row-1)
          ei[i][j]=(float) (image1[i][j]-image1[i-1][j])/ratio;
        else
          ei[i][j]=0.5*(image1[i+1][j]-image1[i-1][j])/ratio;
      };
};

for (i=0;i<bon;i++)
  {
    for (j=0;j<bon;j++)
      {
          error[i][j]=get_uv(i,j);
      };
  };


for (i=0;i<bon;i++)
  {
    for (j=0;j<bon;j++)
      {
        space[i][j]=smooth_u(i, j, bon);
      };
  };

for (i=0;i<bon;i++)
  {
    for (j=0;j<bon;j++)
      {
        u[i][j]=space[i][j];
        space[i][j]=smooth_v(i, j, bon);
      };
  };


for (i=0;i<bon;i++)
  {
    for (j=0;j<bon;j++)
      {
        v[i][j]=space[i][j];
      };
  };
```

```
if (level==(No_Level-2)) last=1;
for (i=0;i<bon;i++)
  {
    for (j=0;j<bon;j++)
      {
        if (class[i][j].fixed==0)
          {
            Recheck=0;
            dist=distance2(u[i][j],v[i][j],class[i][j].link);
            if (dist==0)
              Recheck=0;
            else if (dist>=margin)
              Recheck=1;
            else
              {
                prob=(margin-dist)/margin;
                if (0) printf ("Prob=%f\n",prob);
                if (prob<0)
                  Abort ("Probability must be greater than zero.");
                else if (nn_rand()>prob) Recheck=1;
              };
            if (Recheck)
              {
                (class[i][j].link->no)--;
                class[i][j].link=
                  checkfeature(u[i][j],v[i][j],head,nofeat,margin, last);
                (class[i][j].link->no)++;
              };
          };
      };
  };
head=tidyfeature(head);

if (DEBUG) printf("Adjust feature statistics\n");
temp=head;
for (k=0; k<nofeature; k++)
  {
    if (0) printf ("k=%d\n", k);
    nonode=temp->no;
    acount=nonode;
    u_ave=0;
    v_ave=0;
    for (i=0;i<bon;i++)
      for (j=0;j<bon;j++)
        {
          if (class[i][j].link==temp)
            {
              u_ave=u_ave+u[i][j];
              v_ave=v_ave+v[i][j];
              acount--;
              if (acount<0) Abort ("Node number too small.");
            };
        };
    if (acount>0) Abort ("Node number too large.");
    u_ave=u_ave/nonode;
    v_ave=v_ave/nonode;

    temp->u=u_ave;
    temp->v=v_ave;
    temp=temp->link;
  };

margin=margin+margin_step;

/* Fix */
```



```
  if (level > (Start_Fix+2))
    {
      for (i=0; i<=bon-1;i++)
        for(j=0; j<=bon-1; j++)
          if (error[i][j]<=fix_threshold)  class[i][j].fixed=1;
      /* fix_threshold=fix_threshold*4; */
    }
};

temp=head;
if (DEBUG) printf("Assign Colour\n");
assigncolour(temp);

if (DEBUG) printf("Re-Size\n");

wlevel=1;
arow=2;

do {
arow=arow*2;
wlevel++;
}
while(arow<bon);

/* printf("level=%d; wlevel=%d", level, wlevel); */
/* wlevel=level-1 */

ratio=power(2,(No_Level-wlevel));

if (0) printf("Ratio=%d\n", ratio);

if (ratio!=1)
  {
    for (i=Row256-1; i>=0; i--)
      {
        for (j=Col256-1; j>=0; j--)
          {
            hi=i/ratio;
            hj=j/ratio;
            class_temp=class[hi][hj];
            class[i][j]=class_temp;
          };
      };
  };

if (0) printf("Produce Segmentation Map\n");

OBuff = (unsigned char *) malloc (Row256 * Col256 * sizeof (char));

if (OBuff==NULL) Abort ("Cannot allocate image buffer.");

obuff=OBuff;
for (i=0; i<Row256; i++)
{
  for (j=0; j<Col256; j++)
    {
      value=class[i][j].link->colour;
      *obuff = (unsigned char) value;
      obuff++;
    }
}

if (0) printf("Output\n");


if ((fd3 = creat(argv[3], 0644)) == -1)
```

```
  Abort ("could not create output file");

whead(fd3, Row256, Col256);

obuff=OBuff;
FWrite(fd3, obuff, Row256*Col256);
close(fd3);

if (no_fixmap==0)
{
  obuff=OBuff;
  for (i=0; i<Row256; i++)
    {
      for (j=0; j<Col256; j++)
        {
          value=(class[i][j].fixed)*200;
          *obuff = (unsigned char) value;
          obuff++;
        }
    }

  strcpy(ss,argv[3]);
  strcat(ss,".fix");
  if ((fd3 = creat(ss, 0644)) == -1)
    Abort ("could not create fixed map");

  whead(fd3, Row256, Col256);

  obuff=OBuff;
  FWrite(fd3, obuff, Row256*Col256);

  close(fd3);
};
Abort ("");
};                                  /* main */


float get_uv(int i, int j)
{
  float b[2];
  float z[2][2];
  float a[4][2];
  float c[4];
  float s1, s2, s3, s4;
  float lsq_error;

  lsq_error=0;

  a[0][0]=ei[2*i][2*j];
  a[1][0]=ei[2*i+1][2*j];
  a[2][0]=ei[2*i][2*j+1];
  a[3][0]=ei[2*i+1][2*j+1];
  a[0][1]=ej[2*i][2*j];
  a[1][1]=ej[2*i+1][2*j];
  a[2][1]=ej[2*i][2*j+1];
  a[3][1]=ej[2*i+1][2*j+1];

  c[0]=et[2*i][2*j];
  c[1]=et[2*i+1][2*j];
  c[2]=et[2*i][2*j+1];
  c[3]=et[2*i+1][2*j+1];


  z[0][0]=a[0][0]*a[0][0]+a[1][0]*a[1][0]+a[2][0]*a[2][0]+a[3][0]*a[3][0];
  z[0][1]=a[0][0]*a[0][1]+a[1][0]*a[1][1]+a[2][0]*a[2][1]+a[3][0]*a[3][1];
  z[1][0]=z[0][1];
```



```
  z[1][1]=a[0][1]*a[0][1]+a[1][1]*a[1][1]+a[2][1]*a[2][1]+a[3][1]*a[3][1];

  /*printf ("z00=%f\n",z[0][0]);
  printf ("z01=%f\n",z[0][1]);
  printf ("z10=%f\n",z[1][0]);
  printf ("z11=%f\n",z[1][1]);*/

  b[0]=a[0][0]*c[0]+a[1][0]*c[1]+a[2][0]*c[2]+a[3][0]*c[3];
  b[1]=a[0][1]*c[0]+a[1][1]*c[1]+a[2][1]*c[2]+a[3][1]*c[3];

  /* printf ("b0=%f\n",b[0]);
  printf ("b1=%f\n",b[1]); */

  if (z[0][0]!=0)
    {
      if (z[1][1]==0)
        {
          u[i][j]=b[0]/z[0][0];
          v[i][j]=0;
        }
      else
        {
          z[1][1]=z[1][1]-(z[1][0]*z[0][1]/z[0][0]);
          if (z[1][1]==0)
            {
              u[i][j]=z[0][0]*b[0]/(z[0][0]*z[0][0]+z[0][1]*z[0][1]);
              v[i][j]=z[0][1]*b[0]/(z[0][0]*z[0][0]+z[0][1]*z[0][1]);
            }
          else
            {
              b[1]=b[1]-(z[1][0]*b[0]/z[0][0]);
              v[i][j]=b[1]/z[1][1];
              u[i][j]=(b[0]-z[0][1]*v[i][j])/z[0][0];
            };
        };
    }
  else
    {
      if (z[1][1]!=0)
        {
          u[i][j]=0;
          v[i][j]=b[1]/z[1][1];
        }
      else
        {
          u[i][j]=v[i][j]=0;
          lsq_error=EXTREME;
        }
    };

  if (lsq_error!=EXTREME)
    {
      s1=(a[0][0]+u[i][j]+a[0][1]*v[i][j]-c[0]);
      s2=(a[1][0]+u[i][j]+a[1][1]*v[i][j]-c[1]);
      s3=(a[2][0]+u[i][j]+a[2][1]*v[i][j]-c[2]);
      s4=(a[3][0]+u[i][j]+a[3][1]*v[i][j]-c[3]);

      lsq_error=s1*s1+s2*s2+s3*s3+s4*s4;
    };
  return (lsq_error);
}

float smooth_u(int i, int j, int bon)
{

  float temp;
```

```
   int count, ki, kj;

   count=0;
   temp=0;
   for (ki=0; ki<=0; ki++)
     for (kj=0; kj<=0; kj++)
       {
         if ((i+ki)>=0 && (i+ki)<bon && (j+kj)>=0 &&(j+kj)<bon)
           temp=temp+u[i+ki][j+kj];
         count++;
       };

  temp=temp/count;

  return(temp);
};

float smooth_v(int i, int j, int bon)
{

  float temp;
  int count, ki, kj;

  count=0;
  temp=0;
  for (ki=0; ki<=0; ki++)
    for (kj=0; kj<=0; kj++)
      {
        if ((i+ki)>=0 && (i+ki)<bon && (j+kj)>=0 &&(j+kj)<bon)
          temp=temp+v[i+ki][j+kj];
        count++;
      };

  temp=temp/count;

  return(temp);
};
```

```
# Definitions section

CC=gcc
INCLUDE=-I.
CFLAGS=-g $(INCLUDE)
ARFLAGS = rv
AR=ar
CP=cp
RM=/bin/rm

# Main Body

stedge: stedge.o rasterio.o
        $(CC) $(CFLAGS) -o stedge stedge.o rasterio.o -lm

of:     of.o rasterio.o mof.o stats.o
        $(CC) $(CFLAGS) -o of of.o rasterio.o mof.o stats.o -lm

of2:    of2.o rasterio.o mof.o stats.o
        $(CC) $(CFLAGS) -o of2 of2.o rasterio.o mof.o stats.o -lm

pyd: pyd.o rasterio.o
        $(CC) $(CFLAGS) -o  pyd rasterio.o pyd.o -lm

genof: genof.o rasterio.o
        $(CC) $(CFLAGS) -o  genof rasterio.o genof.o -lm

genof2: genof2.o rasterio.o
        $(CC) $(CFLAGS) -o  genof2 rasterio.o genof2.o -lm

combine: combine.o rasterio.o
        $(CC) $(CFLAGS) -o  combine rasterio.o combine.o -lm
```

```
struct feature {
float u;
float v;
int no;
int colour;
struct feature *link;
};

struct class_table {
struct feature *link;
int fixed;
};

extern int nofeature;


#ifndef _MTS_H
#define _MTS_H


void assigncolour(struct feature *pointer);
void quadruple_feature(struct feature *pointer);
struct feature *checkfeature(float d1, float d2, struct feature *fhead, int nofeat, f
loat margin, int last);
float distance2(float d1, float d2, struct feature *pointer);
struct feature *tidyfeature(struct feature *ptr);
#endif
```

```
#include <stdio.h>
#include "mof.h"
#include "rasterio.h"


void assigncolour(struct feature *pointer)
{

struct feature *ptr, *temp;
int colour, cjump, cjump2, iter;

cjump=255/nofeature;
cjump2=cjump/3;

iter=nofeature;
ptr=pointer;
temp=ptr;
colour=0;

do {
colour=colour+2*cjump2;
if (1) printf("Colour=%d\n",colour);
if (1) printf("No=%d\n",temp->no);
if (1) printf("u=%f\n",temp->u);
if (1) printf("v=%f\n",temp->v);
temp->colour=colour;
temp=(temp->link);
iter--;
if (iter<0) Abort ("Feature table is too long (assign colour).");
} while(temp!=NULL);

if (iter>0) Abort ("Feature table is too short (assign colour).");
};



void quadruple_feature(struct feature *pointer)
{
struct feature *temp;
int iter;

iter=nofeature;
temp=pointer;
do {
temp->no=4*(temp->no);
if (0) printf("No=%d\n", temp->no);

temp=temp->link;
iter--;
if (iter<0) Abort ("Feature table is too long.");
} while(temp!=NULL);

if (iter>0) Abort ("Feature table is too short.");
};


struct feature *checkfeature(float d1, float d2, struct feature *fhead, int nofeat, f
loat margin, int last)
{
int iter, counter;
float adist, mindist;
struct feature *temp, *temp2, *temp3;

iter=nofeature;
temp=fhead;
```



```
counter=0;
mindist=100000000;

if (0) printf("No. of Feature=%d\n",nofeature);
do {

adist=distance2(d1, d2, temp);
if (DEBUG) printf ("Dist=%f\n",adist);
if (adist<margin || last==1)
  {
    counter++;
    if (adist < mindist)
      {
        temp2=temp;
        mindist=adist;
      }
    else
      {
        if (DEBUG) printf("Adist(%f) > Mindist(%f)\n", adist,mindist);
      };
  };
temp3=temp;
temp=temp->link;
iter--;
if (iter<0) Abort ("Feature table is too long (check feature).");
} while(temp!=NULL);

if (iter>0) Abort ("Feature table is too short (check feature).");

/* if (counter>7) Abort ("Margin too large."); */

if (counter==0)
  {
    temp2=(struct feature *) malloc (sizeof (struct feature));
    temp3->link=temp2;
    temp2->u=d1;
    temp2->v=d2;
    temp2->no=0;
    temp2->link=NULL;
    nofeature++;
  };
if (nofeature>nofeat) Abort ("Two many features detected.");
return (temp2);
};



float distance2(float d1, float d2, struct feature *pointer)
{
float adist;
float value, value2;
struct feature *ptr;

ptr=pointer;

value=d1-(ptr->u);
value2=d2-(ptr->v);
adist= (float) sqrt(value*value+value2*value2);

return (adist);
};


struct feature *tidyfeature(struct feature *ptr)
{
int iter;
```

```
struct feature *ahead, *temp, *temp2;

temp=ptr;
while (temp->no<=0)
  {
    if (1) printf("Feature head cut off\n");
    if (0) printf("No=%d\n", temp->no);
    if (temp->no<0) Abort ("Wrong node number.");
    temp=temp->link;
    nofeature--;
  };

ahead=temp;
if (0) printf("No=%d\n", temp->no);

temp2=temp->link;
while(temp2!=NULL)
{

  if (temp2->no<=0)
    {
      if (1) printf("Feature table cut off\n");
      if (1) printf("No=%d\n", temp2->no);

      if (temp->no<0) Abort ("Wrong node number.");

      temp->link=temp2->link;
      temp2=temp2->link;
      nofeature--;
    }
  else
    {
      temp=temp2;
      temp2=temp2->link;
    };
};
if (1) printf("No. of Feature=%d\n",nofeature);
return(ahead);

};
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "rasterio.h"

#define ImagSize   257
#define BUFSIZE    257

char  ProgName[15];

unsigned char   Image[ImagSize][ImagSize],temp[ImagSize][ImagSize];
unsigned char   17[128][128];
unsigned char   16[64][64];
unsigned char   15[32][32];
unsigned char   14[16][16];
unsigned char   13[8][8];
unsigned char   12[4][4];


int Row1,                       /* # rows in image              */
    Col1;                       /* # columns in image           */

void Usage()
{
fprintf (stderr, "Command: %s Input-image\n", ProgName);
}

void main(argc,argv)
int  argc;
char *argv[];
{

char ss[20];

int fd1, fd2, fd3, fd4, fd5, fd6, fd7;
int i,j;
unsigned char buf[BUFSIZE];
unsigned char *ptr;
float filter[]={0.05,0.25,0.4,0.25,0.05};

strcpy (ProgName, argv[0]);
if (argc != 2)
  {
    Usage();
    Abort("");
  };

if ((fd1 = open(argv[1], 0644)) == -1)
  Abort ("could not open input file.");

if(0) printf("%f\n", filter[0]);
if(0) printf("%f\n", filter[1]);
if(0) printf("%f\n", filter[2]);
if(0) printf("%f\n", filter[3]);
if(0) printf("%f\n", filter[4]);


/* read in initial image header */

rhead(fd1, &Row1, &Col1);

for (i=0; i<Row1;i++)
  {
    FRead(fd1,buf,Col1);
    for (j=0; j < Col1; j++)
      Image[i][j] = buf[j];
  };

close(fd1);

/* create level 7*/

for (i=0; i<Row1; i++)
  {
    temp[i][0]=(filter[2]*Image[i][0]+filter[3]*Image[i][1]
       +filter[4]*Image[i][2])*10/7;

    for (j=1; j <127; j++)
      temp[i][j] = filter[0]*Image[i][2*j-2]+filter[1]*Image[i][2*j-1]
         +filter[2]*Image[i][2*j]+filter[3]*Image[i][2*j+1]
         +filter[4]*Image[i][2*j+2];

    temp[i][127]=(filter[0]*Image[i][252]+filter[1]*Image[i][253]
       +filter[2]*Image[i][254]+filter[3]*Image[i][255])*100/95;
  };

for (j=0; j<128;j++)
  {
    17[0][j]=(filter[2]*temp[0][j]+filter[3]*temp[1][j]
       +filter[4]*temp[2][j])*10/7;

    for (i=1; i<127; i++)
      17[i][j] = filter[0]*temp[2*i-2][j]+filter[1]*temp[2*i-1][j]
         +filter[2]*temp[2*i][j]+filter[3]*temp[2*i+1][j]
         +filter[4]*temp[2*i+2][j];

    17[127][j]=(filter[0]*temp[252][j]+filter[1]*temp[253][j]
       +filter[2]*temp[254][j]+filter[3]*temp[255][j])*100/95;

  };

strcpy(ss, argv[1]);
strcat(ss, ".17");

if ((fd7 = creat(ss, 0644)) == -1)
  Abort ("could not create output file.");

whead(fd7, 128, 128);

ptr=&17[0][0];
FWrite(fd7, ptr, 128*128);

close(fd7);

/* create level 6*/

for (i=0; i<128; i++)
  {
    temp[i][0]=(filter[2]*17[i][0]+filter[3]*17[i][1]
       +filter[4]*17[i][2])*10/7;

    for (j=1; j <63; j++)
      temp[i][j] = filter[0]*17[i][2*j-2]+filter[1]*17[i][2*j-1]
         +filter[2]*17[i][2*j]+filter[3]*17[i][2*j+1]
         +filter[4]*17[i][2*j+2];

    temp[i][63]=(filter[0]*17[i][124]+filter[1]*17[i][125]
       +filter[2]*17[i][126]+filter[3]*17[i][127])*100/95;
  };

for (j=0; j<64;j++)
  {
```

```
    16[0][j]=(filter[2]*temp[0][j]+filter[3]*temp[1][j]
       +filter[4]*temp[2][j])*10/7;

    for (i=1; i<63; i++)
      16[i][j] = filter[0]*temp[2*i-2][j]+filter[1]*temp[2*i-1][j]
         +filter[2]*temp[2*i][j]+filter[3]*temp[2*i+1][j]
         +filter[4]*temp[2*i+2][j];

    16[63][j]=(filter[0]*temp[124][j]+filter[1]*temp[125][j]
       +filter[2]*temp[126][j]+filter[3]*temp[127][j])*100/95;

  };

strcpy(ss, argv[1]);
strcat(ss, ".16");

if ((fd6 = creat(ss, 0644)) == -1)
  Abort ("could not create output file.");

whead(fd6, 64, 64);

ptr=&16[0][0];
FWrite(fd6, ptr, 64*64);

close(fd6);

/* create level 5*/

for (i=0; i<64; i++)
  {
    temp[i][0]=(filter[2]*16[i][0]+filter[3]*16[i][1]
       +filter[4]*16[i][2])*10/7;

    for (j=1; j <31; j++)
      temp[i][j] = filter[0]*16[i][2*j-2]+filter[1]*16[i][2*j-1]
         +filter[2]*16[i][2*j]+filter[3]*16[i][2*j+1]
         +filter[4]*16[i][2*j+2];

    temp[i][31]=(filter[0]*16[i][60]+filter[1]*16[i][61]
       +filter[2]*16[i][62]+filter[3]*16[i][63])*100/95;
  };

for (j=0; j<32;j++)
  {
    15[0][j]=(filter[2]*temp[0][j]+filter[3]*temp[1][j]
       +filter[4]*temp[2][j])*10/7;

    for (i=1; i<31; i++)
      15[i][j] = filter[0]*temp[2*i-2][j]+filter[1]*temp[2*i-1][j]
         +filter[2]*temp[2*i][j]+filter[3]*temp[2*i+1][j]
         +filter[4]*temp[2*i+2][j];

    15[31][j]=(filter[0]*temp[60][j]+filter[1]*temp[61][j]
       +filter[2]*temp[62][j]+filter[3]*temp[63][j])*100/95;

  };

strcpy(ss, argv[1]);
strcat(ss, ".15");

if ((fd5 = creat(ss, 0644)) == -1)
  Abort ("could not create output file.");

whead(fd5, 32, 32);

ptr=&15[0][0];
```



```
FWrite(fd5, ptr, 32*32);

close(fd5);

/* create level 4*/

for (i=0; i<32; i++)
  {
    temp[i][0]=(filter[2]*15[i][0]+filter[3]*15[i][1]
       +filter[4]*15[i][2])*10/7;

    for (j=1; j <15; j++)
      temp[i][j] = filter[0]*15[i][2*j-2]+filter[1]*15[i][2*j-1]
         +filter[2]*15[i][2*j]+filter[3]*15[i][2*j+1]
         +filter[4]*15[i][2*j+2];

    temp[i][15]=(filter[0]*15[i][28]+filter[1]*15[i][29]
       +filter[2]*15[i][30]+filter[3]*15[i][31])*100/95;
  };

for (j=0; j<16;j++)
  {
    14[0][j]=(filter[2]*temp[0][j]+filter[3]*temp[1][j]
       +filter[4]*temp[2][j])*10/7;

    for (i=1; i<15; i++)
      14[i][j] = filter[0]*temp[2*i-2][j]+filter[1]*temp[2*i-1][j]
         +filter[2]*temp[2*i][j]+filter[3]*temp[2*i+1][j]
         +filter[4]*temp[2*i+2][j];

    14[15][j]=(filter[0]*temp[28][j]+filter[1]*temp[29][j]
       +filter[2]*temp[30][j]+filter[3]*temp[31][j])*100/95;

  };

strcpy(ss, argv[1]);
strcat(ss, ".14");

if ((fd4 = creat(ss, 0644)) == -1)
  Abort ("could not create output file.");

whead(fd4, 16, 16);

ptr=&14[0][0];
FWrite(fd4, ptr, 16*16);

close(fd4);

/* create level 3*/

for (i=0; i<16; i++)
  {
    temp[i][0]=(filter[2]*14[i][0]+filter[3]*14[i][1]
       +filter[4]*14[i][2])*10/7;

    for (j=1; j <7; j++)
      temp[i][j] = filter[0]*14[i][2*j-2]+filter[1]*14[i][2*j-1]
         +filter[2]*14[i][2*j]+filter[3]*14[i][2*j+1]
         +filter[4]*14[i][2*j+2];

    temp[i][7]=(filter[0]*14[i][12]+filter[1]*14[i][13]
       +filter[2]*14[i][14]+filter[3]*14[i][15])*100/95;
  };

for (j=0; j<8;j++)
  {
```

```
    l3[0][j]=(filter[2]*temp[0][j]+filter[3]*temp[1][j]
       +filter[4]*temp[2][j])*10/7;

    for (i=1; i<7; i++)
      l3[i][j] = filter[0]*temp[2*i-2][j]+filter[1]*temp[2*i-1][j]
         +filter[2]*temp[2*i][j]+filter[3]*temp[2*i+1][j]
         +filter[4]*temp[2*i+2][j];

    l3[7][j]=(filter[0]*temp[12][j]+filter[1]*temp[13][j]
       +filter[2]*temp[14][j]+filter[3]*temp[15][j])*100/95;

  };

strcpy(ss, argv[1]);
strcat(ss, ".l3");

if ((fd3 = creat(ss, 0644)) == -1)
  Abort ("could not create output file.");

whead(fd3, 8, 8);

ptr=&l3[0][0];
FWrite(fd3, ptr, 8*8);

close(fd3);

Abort ("");
}                                /* main */
```

```
#ifndef _RASTERIO_H
#define _RASTERIO_H

void Abort(char   *string);
void whead(int fd, int rows, int cols);
void rhead(int fd, int *pRows, int *pCols);
void rhead(int fd, int *pRows, int *pCols);
void outint(int fd, int m);
void FRead(int fd, unsigned char *pointer, int numbytes);
void FWrite(int fd, unsigned char *pointer, int len);
void RScale(float *fbuff, unsigned char *cbuff, int numpix);

#endif

#define DEBUG 0
```

```
#include <stdio.h>
#include <math.h>
#include "rasterio.h"

void Abort(char *string)
{
    if (*string)
        fprintf (stderr, "ERROR: %s\n", string);

    exit (3);
}                               /* Abort */

void whead(int fd, int rows, int cols)
{
  outint(fd,1504078485);
  outint(fd,cols);
  outint(fd,rows);
  outint(fd,8);
  outint(fd,cols*rows);
  outint(fd,1);
  outint(fd,2);
  outint(fd,0);
}

void rhead(int fd, int *pRows, int *pCols)
{
        unsigned char buf[32];
        int length;

        if (read(fd, buf, 32)!=32)
                abort("Read Image header error.");

        *pCols=buf[4]*256*256*256+buf[5]*256*256+buf[6]*256+buf[7];
        *pRows=buf[8]*256*256*256+buf[9]*256*256+buf[10]*256+buf[11];
        length=buf[28]*256*256*256+buf[29]*256*256+buf[30]*256+buf[31];
        if (0) printf("\nMaplength = %d\n", length);
        if (lseek (fd, length,1)==-1) Abort ("lseek error");
}

void outint(int fd, int m)
{ unsigned char buf[4];
  int n1,n2,n3,i;
  n1=256;
  n2=256*256;
  n3=256*256*256;
  buf[0]=m/n3;
  buf[1]=(m-buf[0]*n3)/n2;
  buf[2]=(m-buf[0]*n3-buf[1]*n2)/n1;
  buf[3]=m-buf[0]*n3-buf[1]*n2-buf[2]*n1;
  if (write(fd,buf,4)!=4) abort("writing error\n");
}

 /*
  * FRead - read 'numbytes' to 'pointer' and check for errors.
  */


void FRead(int fd, unsigned char *pointer, int numbytes)
{

  if (read (fd, pointer, numbytes) != numbytes)
    Abort ("reached end of input file");

}                               /* FRead */
```



```
 /*
  * FWrite - write 'len' bytes from 'pointer' to stdout, checking for errors.
  */

void FWrite(int fd, unsigned char *pointer, int len)
{
    if (write (fd, pointer, len) != len)
        Abort ("cannot write to output");
}                               /* FWrite */


 /*
  * RScale: Re-scale floating buffer so it is between 0 -- 255.
  */

void RScale(float *fbuff, unsigned char *cbuff, int numpix)
{
  int n;
  float   max = 0.0, val, value;
  float scale;
  float *fbuffptr;
  unsigned char *cbuffptr;

  n=numpix;
  fbuffptr=fbuff;
  do {
    val = *(fbuffptr++);
    if (val > max)
      max = val;
  } while (--n);
  if (1) printf("Max=%f\n",max);
  scale =255/max;
  n=numpix;
  fbuffptr=fbuff;
  cbuffptr=cbuff;

  do
    {
      value = (*(fbuffptr++) * scale);
      if(value>255)
        *(cbuffptr++)=255;
      else if (value<0)
        *(cbuffptr++)=0;
      else
        *(cbuffptr++)=(unsigned char)value;
    }
  while (--n);

}
```

```
#include <sys/time.h>

#define MAXINT  0x7FFF
#define FALSE 0
#define TRUE 1

static char     RandFlag = FALSE;       /* true if 2nd rnd # generated  */

static float    Xx1,                    /* used by the 'NRand' routine  */
                Xx2;

static struct timeval   tv;
static struct timezone  tz;

#ifndef _STATES_H
#define _STATES_H

float image_mean(unsigned char *imageptr, int sizex, int sizey);
float image_var(unsigned char *imageptr, int sizex, int sizey, float mean);
float image_skew(unsigned char *imageptr, int sizex, int sizey, float mean);
float getmean(unsigned char *dataptr, int startx,int starty,int endx,int endy, int si
zex);
float variance (unsigned char *dataptr,int startx,int starty,int endx,int endy, int s
izex, float mean);
float skewness (unsigned char *dataptr,int startx,int starty,int endx,int endy, int s
izex, float mean);
void test(unsigned char *imageptr, int sizex, int sizey);
int power( int base, int n);
void gen_seed(void);
float n_rand(void);
float nn_rand(void);

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "rasterio.h"
#include "stats.h"

int power( int base, int n)
{
int i,p;

p=1;
for(i=1;i<=n;i++)
  p=p*base;

return p;
}


float image_mean(unsigned char *imageptr, int sizex, int sizey )
{
        float mean;
        int i,j;

        mean = 0;

        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
              mean = mean + *imageptr;
              imageptr++;
            };
        mean = mean / (sizex*sizey);

        if (DEBUG) printf("Image mean : %f\n", mean);

        return(mean);
}

float image_var(unsigned char *imageptr, int sizex, int sizey, float mean)
{
        float sum, var;
        int i,j;

        var = 0; sum = 0;
        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
              sum = sum + (*imageptr-mean)*(*imageptr-mean);
              imageptr++;
            };
        var = sum / (sizex*sizey);

        if (DEBUG) printf("Image variance : %f\n", var);

        return(var);
}

float image_skew(unsigned char *imageptr, int sizex, int sizey, float mean)
{
        float sum, skew;
        int i,j;

        skew = 0; sum = 0;
        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
```



```
              sum = sum
                + (*imageptr-mean)*(*imageptr-mean)*(*imageptr-mean);
              imageptr++;
            };
        skew = sum/(sizex*sizey);

        if (DEBUG) printf("Image skew : %f\n", skew);

        return(skew);
}

float getmean(unsigned char *dataptr, int startx,int starty,int endx,int endy, int si
zex)
{
        float mean;
        int i,j, mean2;
        unsigned char *ptr;

        mean=0.0;
        mean2=0;

        if (DEBUG) printf("value : %i\n", *dataptr);

        for (i=startx; i<=endx; i++)
                for (j=starty; j<=endy; j++)
                  {
                    ptr=dataptr+(i*sizex+j);
                    if (0) printf("value : %i\n", (int)*ptr);
                    mean2 = mean2 + (int) *ptr;
                  }
        if (DEBUG) printf("Image mean2 : %i\n", mean2);
        mean = (float) mean2/ (float)((endx-startx+1)*(endy-starty+1));
        if (DEBUG) printf("startx=%i, endx=%i, starty=%i, endy=%i\n", startx,endx,sta
rty,endy);

        if (DEBUG) printf("Image mean : %f\n", mean);

        return(mean);
}

float variance (unsigned char *dataptr,int startx,int starty,int endx,int endy, int s
izex, float mean)
{
        float var, sum, value;
        int i,j;
        unsigned char *ptr;
        if (DEBUG) printf(" %d %d %d %d \n", startx, starty, endx, endy);
        var = 0; sum =0;

        for (i=startx; i<=endx; i++)
          for (j=starty; j<=endy; j++)
            {
              ptr=dataptr+(i*sizex+j);
              value=(float)*ptr;
              sum = sum + (value - mean)*(value - mean)  ;
            }
        var = (float) sum / (float)((endx-startx+1) * (endy-starty+1) );

        return (var);
}

float skewness (unsigned char *dataptr,int startx,int starty,int endx,int endy, int s
izex, float mean)
{
        float skew, sum, value;
        int i,j;
```

```
        unsigned char *ptr;
        if (DEBUG) printf(" %d %d %d %d \n", startx, starty, endx, endy);
        skew = 0; sum =0;

        for (i=startx; i<=endx; i++)
          for (j=starty; j<=endy; j++)
            {
              ptr=dataptr+(i*sizex+j);
              value=(float)*ptr;
              sum = sum + (value - mean)*(value - mean)*(value - mean) ;
            }
        skew = (float) sum / (float)((endx-startx+1)*(endy-starty+1));

        return (skew);
}

void test(unsigned char *imageptr, int sizex, int sizey)
{
        int i,j;
        unsigned char *ptr;

        ptr=imageptr;
        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
              if (1) printf("Image value : %i\n", *ptr);
              ptr++;
            }
}

void test2(float *imageptr, int sizex, int sizey)
{
        int i,j;
        float *ptr;

        ptr=imageptr;
        for (i=0; i<=sizex-1; i++)
          for (j=0; j<=sizey-1; j++)
            {
              if (1) printf("Image value : %8.3f\n", *ptr);
              ptr++;
            }
}


void gen_seed(void)
{
  gettimeofday (&tv, &tz);      /* initialize random seed */
  srand (getpid() ^ tv.tv_sec ^ tv.tv_usec);
}

float n_rand(void)
{       float v1, v2, s, temp1, temp2;

        if (RandFlag) {  /* if RandFlag, then return 2nd random number */
                RandFlag = FALSE;
                if (0) printf("Rand2=%f\n",Xx2);
                return (Xx2);
        }
        RandFlag = TRUE;

       /*
        * The following code generates 2 Standard Normally distributed numbers
        * into 'Xx1' and 'Xx2' using the Polar method for normal deviates.
        */
```



```
        /*printf("start_rand\n");*/
        do {
            v1 = (double) (rand () & MAXINT) / (double) MAXINT * 2.0 - 1.0;
            v2 = (double) (rand () & MAXINT) / (double) MAXINT * 2.0 - 1.0;
            s = v1 * v1 + v2 * v2;
            /*printf("s=%f\n",s);*/

         ) while (s >= 1.0);

        Xx1 = v1 * sqrt ((-2.0 * log (s)) / s);
        Xx2 = v2 * sqrt ((-2.0 * log (s)) / s);

        if (0) printf("Rand1=%f\n",Xx1);
        return (Xx1);
}

float nn_rand(void)
{
  float temp;

  do
    temp=n_rand();
  while ((temp<-0.5) ||(temp >0.5));

  temp=temp+0.5;
  return(temp);
}
```

Bibliography

[1] J.Ahlberg, E.Nilson and J.Walsh, *The Theory of Splines and Their Applications, in Mathematics in Science and Engineering*, vol. 38, New York: Academic, 1967.

[2] P.Andrey and P.Tarroux, "Unsupervised texture segmentation using selectionist relaxation," in *Proc. 4th European Conference on Computer Vision (ECCV'96)*, vol.1, pp.482-491, 1996.

[3] J.Babaud, A.Witkin, M.Baudin and R.Duda, "Uniqueness of the Gaussian kernel for scale-space filtering," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-8, pp.26-33, 1986.

[4] J.L. Barron, D.J. Fleet and S.S. Beauchemin, "Systems and experiment performance of optical flow techniques," *Int. J. Comput. Vision,* vol.12, pp.43-77, 1994.

[5] F.Bergholm, "Edge focusing," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-9, pp.726-741, 1987.

[6] M.Bertero, T.Poggio and V.Torre, "Ill-posed problems in early vision," in *Proc. IEEE*, vol. 76, pp.869-889, 1988.

[7] P.J.Besl and R.C.Jain, "Segmentation through Variable-order Surface Fitting," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-10, pp. 167-192, 1988.

[8] M.J.Black and A.D.Jepson, "EigenTracking: robust matching and tracking of articulated objects using a view-based representation," in *Proc. 4th European Conference on Computer Vision (ECCV'96),* vol.1, pp. 329-342, 1996.

[9] A.Blake and A.Yuille, *Active vision,* MA: MIT Press, 1993.

[10] P. Bouthemy and E. Francois, "Motion segmentation and qualitative dynamic scene analysis from an image sequences," *Int. J. Comput. Vision,* vol.10, pp.157-182, 1993.

[11] C.Bouman and B.Liu, "Multiple resolution segmentation of textured images," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-13, pp.99-113, 1991.

[12] C.Bouman and M.Shapiro, "A multiscale random field model for Bayesian image segmentation," *IEEE Trans. Image Process.*, vol. IP-3, pp.162-177, 1994.

[13] P.J. Burt, E.H. Adelson, "The Laplacian Pyramid as a Compact Image Code," *IEEE Trans. Commun.*, vol. COM-31, pp.337-345 , 1983.

[14] A.D.Calway and R.Wilson, "Curve extraction in images using a multiresolution framework," *CVGIP: Image understanding,* vol. 59, pp.359-366, 1994.

[15] T.Chang and C.C.J.Kuo, "Texture Analysis and Classification with Tree-Structured Wavelet Transform," *IEEE Trans. Image Processing,* vol. IP-2, pp.429-441, 1993.

[16] J.Canny, "A computational approach to edge detection," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-8, pp.679-698, 1986.

[17] G.Chen and Y.H.Yang, "Edge detection by regularized cubic B-spline fitting," *IEEE Trans. Syst. Man Cybern.*, vol. SMC-25, pp.636-643, 1995.

[18] Z.Cho, J.P.Jones and M.Singh, *Foundations of medical imaging,* New York: Wiley, 1993.

[19] J.Clark, "Authenticating edge produced by zero-crossing algorithms," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-11, pp.43-57, 1989.

[20] L.Cohen, *Time-frequency analysis,* New Jersey: Prentice-Hall, 1995.

[21] R.W.Conners and C.A.Harlow, "A theoretical comparison of texture algorithms," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-2, pp. 204-222, 1980.

[22] G.R.Cross and A.K.Jain, "Markov random field texture models," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-5, pp. 25-39, 1983.

[23] I.Daubechies, "Orthonormal bases of Compactly Supported Wavelets," *Commun. Pure Appl. Math.*, vol.41, pp.909-996, 1988.

[24] I.Daubechies, "The Wavelet Transform, Time-Frequency Localization and Signal Analysis," *IEEE Trans. Inform. Theory*, vol. IT-36, pp.961-1004, 1990.

[25] J.G.Daugmann, "Pattern and motion vision without Laplacian zero crossings," *J. Opt. Soc. Am.* vol.A5, pp.1142-1148, 1988.

[26] A. DelBimbo, P. Nesi and J.L.C. Sanz, "Optical flow computation using extended constraints," *IEEE Trans. Image Process.*, vol. IP-5, pp.720-739, 1996.

[27] R.C.Dubes and A.K.Jain, "Random field models in image analysis," *J. Appl. Stat.*, vol.16, pp. 131-164, 1989.

[28] B.Dubuisson and M.Masson, "A statistical decision rule with incomplete knowledge about classes" *Patt. Recognit.* vol.26, pp.155-165, 1993.

[29] J.M.Francos, A.Z.Meiri and B.Porat, "A unified texture model based on 2 2-D wold-like decomposition," *IEEE Trans. Signal Process.*, vol. SP-41, pp. 2665-2678, 1993.

[30] W.Frei and C.C.Chen, "Fast boundary detection: A generalization and a new algorithm," *IEEE Trans. Comput.*, vol. C-26, pp. 988-998, 1977.

[31] K.S.Fu (Ed.), *Digital pattern recognition*, Germany: Springer-Verlag, 1976.

[32] D.Geiger and T.Poggio, "An optimal scale for edge detection," in *Proc. IJCAI'87*, Milan, Italy, pp.745-748, 1987.

[33] S.Geman and D.Geman, "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-6, pp. 721-741, 1984.

[34] D.Geman, S.Geman, C.Graffigne and P.Dong, "Boundary detection by constrained optimization," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-12, pp. 609-628, 1990.

[35] A.Grossman and J.Morlet, "Decomposition of Hardy Functions into Square Integrable Wavelets of Constant Shape," *SIAM J. Appl. Math.*, vol.15, pp.723-726, 1984.

[36] W.E.L.Grimson and E.C.Hildreth, "Comments on Digital step edges from zero crossing of second directional derivatives," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-7, pp.121-127, 1985.

[37] R.M.Haralick ,K.Shanmugan and I.Dinstein, "Texture features for image classification," *IEEE Trans. Syst. Man, Cybern.*, vol. SMC-3, pp. 610-621, 1973.

[38] R.M.Haralick, "Ridges and Valleys on Digital Images," *Comput. Vision Graphics Image Processing*, vol. 22, pp. 28-38, 1983.

[39] R.M.Haralick, "Digital step edges from zero crossing of second directional derivatives," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-6, pp.56-68, 1984.

[40] R.M.Haralick and L.G.Shapiro *Computer and robot vision.*, vol. 1 and 2, Addison-Wesley, 1992.

[41] R.Hoffman and A.K.Jain, "Segmentation and Classification of Range Images," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-9, pp. 608-620, 1987.

[42] T.Hofmann, J.Puzicha and J.M.Buhmann, "Unsupervised Segmentation of Textured Images by Pairwise Data Clustering," in *Proc. Inter. Conf. Image Process.*, Lausanne, vol.3, pp.137-140, 1996.

[43] B.K.P. Horn and B.G. Schunck, "Determining optic flow," *Artificial Intell.*, vol.17, pp.185-203, 1981.

[44] J.Hou and R.Bamberger, "Orientation Selective Operators for Ridge, Valley, Edge and Line Detection in Imagery," in *Proc IEEE ICASSP*, vol. 5, pp. 25-28, 1994.

[45] T.I.Hsu and R.Wilson, "A two-component model of texture for analysis and synthesis," Technical Report RR308, Department of Computer Science, University of Warwick, U.K. 1995.

[46] B.R.Hunt, "Applications of constrained least squares estimation to image restoration by digital computers," *IEEE Trans. Comput.*, vol. C-22, pp. 805-812, 1973.

[47] A.K.Jain, *Fundamentals of digital image processing,* London: Prentice-Hall, 1989.

[48] B.Jelusz, "Textons, the elements of texture perception and their interactions," *Nature,* vol. 290, pp. 91-97, 1981.

[49] M.G.Kang and A.K.Katsaggelos, "General choice of the regularization functional in regularized image restoration," *IEEE Trans. Image Processing,* vol. IP-4, pp.594-602, 1995.

[50] M.Kass, A.Witkin and D.Terzopoulos, "Snakes: Active Contour Models," *Int. J. Comput. Vision*, vol. 1, pp. 321-331, 1988.

[51] J.K. Kearney, W.B. Thompson and D.L. Boley, "Optical flow estimation: An error analysis of gradient-based methods with local optimisation," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-9, pp.229-244, 1987.

[52] S.Kirkpatrick, C.D.Gelatt,Jr. and M.P.Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671-680, 1983.

[53] J.J.Koenderink, "The structure of images," *Biol. Cybern.*, vol. 50, pp.363-370, 1984.

[54] D.Lee, "Coping with discontinuities in computer vision: their detection, classification, and measurement," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-12, pp.321-343, 1990.

[55] C.T.Li, and R.Wilson, 'Textured Image Segmentation Using Multiresolution Markov Fields and a Two-component Texture Model," in *Proc.10th Scandinavian Conf. Image Anal. (SCIA'97)*, 1997.

[56] K.Liang, T.Tjahjadi and Y.H.Yang, "A regularized multiscale edge detection scheme using cubic B-spline," in *Proc. U.K. Symposium Time-Frequency Time-Scale Methods (TFTS'95), IEEE Signal Processing Chapter (UKRI section)*, pp.58-65, 1995.

[57] K.H. Liang, T.Tjahjadi and Y.H. Yang, "Multiscale texture segmentation based on image spectrum," in *Proc. IEEE Nordic Signal Processing Symposium (NORSIG'96)*, pp. 239-242, 1996.

[58] K.H. Liang and T. Tjahjadi, "Texture focusing: A multiresolution approach for segmentation," submitted to *Pattern Recognit.*, awaiting review.

[59] T.Linderberg, "Scale space for discrete signals," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-12, pp.214-254, 1990.

[60] T.Lindeberg and D.Fagerstrom, "Scale-space with causal time direction," in *Proc. 4th European Conference on Computer Vision (ECCV'96)*, vol.1, pp. 229-240, 1996.

[61] J.Liu and Y.H.Yang, "Multiresolution color image segmentation," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-16, pp. 689-700, 1994.

[62] Y.Lu and R.Jain, "Behavior of edges in scale space," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-11, pp.337-356, 1989.

[63] Y.Lu and R.Jain, "Reasoning about edges in scale space," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-14, pp.450-468, 1992.

[64] S.Mallet, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-11, pp.674-693, 1989.

[65] S.Mallet, W.L.Huang, "Singularity Detection and Processing with Wavelets," *IEEE Trans. Inform. Theory*, vol. IT-38, pp.617-643, 1992.

[66] D.Marr and E.Hildreth, "Theory of edge detection," in *Proc. Royal Soc. London Ser. B*, vol. 207, pp.187-217, 1980.

[67] D.Marr, *Vision*, New York: Freeman, 1982.

[68] V.Z.Mesarovic, N.P.Galatsanos and A.K.Katsaggelos, "Regularized Constrained Total Least Squares Image Restoration," *IEEE Trans. Image Processing*, vol. 4, pp. 1096-1108, 1995.

[69] F.G. Meyer and P. Bouthemy, "Region-based tracking using affine motion models in long image sequences," *CVGIP: Image Understanding*, vol.60, pp.119-140, 1994.

[70] O.Monga, N.Armande and P.Montesinos, "Thin Nets and Crest Lines: Application to Satellite Data and Medical Images," in *Proc IEEE ICIP'95*, vol. 2 pp. 468-471, 1995.

[71] R.E.Muzzolini, Y.H.Yang and R.A.Pierson, "Multiresolution texture segmentation with applications to diagnostic ultrasound images," *IEEE Trans. Medical Imaging*, vol. MI-12, pp. 108-123, 1993.

[72] R.E.Muzzolini, Y.H.Yang and R.A.Pierson, "Multidimensional texture characterization," Technical Report, Department of Computational Science, University of Saskatchewan, Canada, 1994.

[73] R.E.Muzzolini, Y.H.Yang and R.A.Pierson, "Texture characterization using robust statistics," *Patt. Recogn.*, vol. , pp.119-128 , 1994.

[74] R.E.Muzzolini, *A volumetric approach to segmentation and texture characterisation of ultrasound images*, PhD thesis, Department of Computational Science, University of Saskatchewan, Canada, 1996.

[75] V.S.Nalwa and T.O.Binford, "On detecting edges," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-8, pp.699-714, 1986.

[76] D. Nesi, A. DelBimbo and D. Ben-Tzvi, "A robust algorithm for optical flow estimation," *CVGIP: Image Understanding,* vol.62, pp.59-68, 1995.

[77] T.Pajdla and V.Halvac, "Surface Discontinuities in Range Images," in *Proc IEEE 4th Int. Conf. Comput. Vision*, pp. 524-528, 1993.

[78] D.Pearson and J.Robinson, "Visual Communication at Very Low Data Rates," in *Proc. IEEE*, vol. 73, pp. 795-812, 1985.

[79] P.Perona and J.Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-12, pp.661-674, 1990.

[80] T.R.Reed, H.Wechsler, "Segmentation of Textured Images and Gestalt Organization Using Spatial/Spatial-Frequency Representations," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-12, pp.1-12, 1990.

[81] T.R.Reed and J.M.Du Buf, "A review of recent texture segmentation and feature extraction techniques," *CVGIP: Image Understanding,* vol. 57, pp.359-372, 1993.

[82] S.J.Reeves and R.M.Mersereau, "Optimal estimation of the regularization parameters and stabilizing functional for regularized image restoration," *Opt. Eng.*, vol. 29, pp.446-454, 1990.

[83] S.Riazanoff, B.Cervelle and J.Chorowicz, "Ridge and Valley Line Extraction from Digital Terrain Models," *Int. J. Remote Sensing* , vol. 9, pp. 1175-1183, 1988.

[84] O.Rioul and M.Vetterli, "Wavelets and signal processing," *IEEE Signal Processing Mag.*, vol. 40, pp.2207-2232, 1991.

[85] A.Rosenfeld and M.Thurston, "Edge and curve detector for visual scene analysis," *IEEE Trans. Comput.*, vol. C-20, pp.562-569, 1971.

[86] A.Rosenfeld, R.A,Hummel and S.W.Zucker, "Scene labelling by relaxation operation," *IEEE Trans. Syst. Man, Cybern.*, vol. SMC-6, pp. 420-433, 1976.

[87] J.A.Schnabel, "Shape Description Methods for Medical Images," Technical Report TR/95/12, Department of Computer Science, University College London, U.K, 1995.

[88] S.M. Smith and J.M. Brady, "ASSET-2: Real-time motion segmentation and shape tracking." *IEEE Trans. Patt. Anal. Mach. Intell.*, vol. PAMI-17 pp.814-820, 1995.

[89] C.Stiller, "Object-based estimation of dense motion fields," *IEEE Trans. Image Process.*, vol. IP-6, pp.234-250, 1997.

[90] G.Strang, *Introduction to Applied Mathematics,* MA: Wellesley-Cambridge Press, 1986.

[91] H.Tamula, S.Mori, and T.Yamawaki, "Texture features corresponding to visual problem," *IEEE Trans. Syst. Man, Cybern.,* vol. SMC-8, pp. 460-473, 1978.

[92] D.Terzopoulos, "Regularization of inverse visual problems involving discontinuities," *IEEE Trans. on Patt. Anal. Machine Intell.*, vol. PAMI-8, pp.413-424, 1986.

[93] A.N.Tikhonov and V.A.Arsenin *Solutions of Ill-posed Problems.* Winston & Sons, Washington, 1977.

[94] V.Torre and T.A.Poggio, "On edge detection," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-8, pp.147-163, 1986.

[95] J.T.Tou and R.C.Gonzalez, *Pattern Recognition Principles,* Massachusetts: Addison-Wesley, 1974.

[96] M.Unser, A.Aldroubi and M.Eden, "The L_2 Polynomial Spline Pyramid," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-15, pp.364-379, 1993.

[97] M.Unser, "Texture Classification and Segmentation Using Wavelet Frames," *IEEE Trans. Image Processing*, vol.**IP-4**, pp.1549-1560, 1995.

[98] M.Vetterli and J.Kovacevic, *Wavelets and subband coding*, Prentice-Hall, 1995.

[99] F.M.Vilnrotter, R.Nevatia and K.E.Price, "Structural analysis of natural textures," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-8, pp.76-89, 1986.

[100] G. Wahba, *Spline Models for Observational Data, CBMS-NSF Regional Conference Series in Applied Mathematics 59, Society for Industrial and Applied Mathematics,* Philadelphia, Pennsylvania, 1990.

[101] J.S.Weszka, C.R.Dyer and A.Rosenfeld, "A comparative study of texture measures for terrain classification," *IEEE Trans. Syst. Man, Cybern.,* vol. SMC-6, pp.269-285, 1976.

[102] R.T.Whitaker and S.M.Pizer, "A multiscale approach to nonuniform diffusion," *CVGIP: Image Understanding,* vol. 57, pp.99-110, 1993.

[103] R.T.Whitaker, "Geometry-limited diffusion," *CVGIP: Image Understanding,* vol. 57, pp.111-120, 1993.

[104] R.Wilson and G.H.Granlund, "The uncertainty principle in image processing," *IEEE Trans. Patt. Anal. Machine Intell.,* vol. PAMI-6, pp. 758-767, 1984.

[105] R.Wilson and M.Spann, *Image segmentation and uncertainty,* Research Studies Press: Britain, 1988.

[106] R.Wilson, A.D.Calway and E.R.S.Pearson, "A generalized wavelet transform for Fourier analysis: the multiresolution Fourier transform and its application to image and audio signal analysis," *IEEE Trans. Inform. theory,* vol. IT-38, pp.674-690, 1992.

[107] R. Wilson, "Wavelets: Why so many varieties," in *Proc. U.K. Symposium on Applications of Time-Frequency Time-Scale Methods,*, pp. 23-32, 1995.

[108] A.Witkin, "Scale-space filtering," in *Proc. 4th. Int. Joint. Conf. on Artificial Intell. (IJCAI),* pp.1019-1022, 1983.

[109] A.Yuille and T.Poggio, "Scaling theorems for zero crossings", *IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-8, pp.15-25, 1986.

[110] D.Ziou and S.Tabbone, "A multi-scale edge detector," *Patt. Recogn.*, vol. 26, pp.1305-1314, 1993.